Automatisiertes Lernen von Neuronalen Netzen zur Objektklassifikation auf heterogenen Systemen

Von der
Carl-Friedrich-Gauß-Fakultät
der Technischen Universität Carolo-Wilhelmina zu Braunschweig

zur Erlangung des Grades eines
Doktoringenieurs (Dr.-Ing.)

genehmigte Dissertation

von
Sönke Michalik
geboren am 23.06.1985
in Goslar

Eingereicht am:	27.08.2018
Disputation am:	23.11.2018
1. Referentin/Referent:	Prof. Dr.-Ing. Mladen Berekovic
2. Referentin/Referent:	Prof. Dr. rer. nat. Jochen Steil

2018

Abstract

Diese Arbeit untersucht den Lernprozess Neuronaler Netze am Beispiel der Objektklassifikation. Hierbei wird sowohl das Training als auch die Ausführung von Neuronalen Netzen im Detail betrachtet, um beide Bereiche zu optimieren. Um den Trainingsprozess zu beschleunigen, wird ein Verfahren zum Automatisierten Lernen von Neuronalen Netzen entwickelt, das unter Angabe von gewünschten Objektklassen eigenständig Datenbanken zum Trainieren Neuronaler Netze aus dem Internet erstellt. Dabei wird ein GPU-basiertes Systemdesign zusammen mit einer Quantisierung in Form von Binären Neuronalen Netzen genutzt, um den Trainingsprozess in einer effizienten automatisierten Umsetzung um den Faktor 1000x zu beschleunigen. Um trotz der reduzierten Datensatzgröße die Trainingsergebnisse der State-of-the-Art Implementierungen zu erreichen, wird hierfür ein adaptives Verfahren zur Anpassung des Lernprozesses eingesetzt. Zusätzlich wurde eine Optimierung der Eingangsdaten durch Einbindung von Tiefeninformationen zur Verbesserung der Klassifikationsergebnisse betrachtet. Für die Ausführung der trainierten Modelle wurde eigens für diese Arbeit ein eingebettetes heterogenes System auf Basis eines Xilinx FPGA SoC entwickelt. Das vorgestellte Kamerasystem bietet hier integrierte Hardwareeinheiten, um Tiefeninformationen in Echtzeit zu berechnen. Für dieses Verfahren wurden eigene Algorithmen zur Tiefenbestimmung aus Stereobildern erforscht und als Hardware-basierte Lösung effizient umgesetzt. Die so bereitgestellten Tiefeninformationen konnten genutzt werden, um die Klassifikation durch Anwendung eines Segmentierungsverfahrens weiter zu verbessern. Dies wurde in Testreihen bestätigt. Für das entwickelte Kamerasystem wurden bestehende Implementierungen Binärer Neuronaler Netze angepasst und um eine eigene Implementierung einer abgestimmten Netzarchitektur erweitert. Hierzu wurden weitreichende Untersuchungen durchgeführt, um eine optimale Architektur für die besonderen Gegebenheiten automatisiert generierter Datensätze zu finden. Das so entstandene AL BNN Framework kann durch Angabe gewünschter Objektklassen diese eigenständig antrainieren und liefert abschließend Modelle mit einer ausgezeichneten Klassifikationsgenauigkeit von 82,3% zur Ausführung auf GPU-basierten Systemen und alternativ auf dem neu entwickelten eingebetteten Kamerasystem “AeonCam”. So können neue Objekte durch das optimierte Trainingsverfahren

des AL BNN Frameworks mit stark reduziertem Zeitaufwand, d.h. unter einer Stunde, antrainiert werden und so die Funktionalität des Kamerasystems für viele Anwendungen der Objektklassifikation flexibel erweitert werden. Dabei konnte die Klassifikation durch die Hardwareunterstützung mit einer Bildwiederholrate von bis zu 108 Bildern pro Sekunde bei einem sehr geringen Energieverbrauch von unter 5W realisiert werden. Dadurch ist das präsentierte System besonders für den Einsatz in mobilen Robotern und Drohnen, bei denen Energieeffizienz eine große Rolle spielt, eine hervorragende Wahl zur Umsetzung einer Objektklassifikation. Hier kann auch die kompakte Bauweise des eingebetteten Systems von 110mm x 75mm x 31mm überzeugen. Diese Arbeit zeigt, wie Automatisiertes Lernen auf heterogenen Systemen angewendet werden kann, damit eingebettete Systeme eigenständig neue Datensätze erstellen und hierdurch die Klassifikation neuer Objekte lernen können.

English Version

This work examines the learning process of neural networks using the example of object classification. Here, the training and execution of neural networks will be considered in detail to optimize both areas. In order to speed up the training process, a method for the automated learning of neural networks is developed, which independently creates databases for training neural networks from the Internet, specifying desired object classes. It uses a GPU-based system design in conjunction with a quantization in the form of binary neural networks to accelerate the training process in an efficient automated implementation by a factor of 1000x. In order to achieve the training results of the state-of-the-art implementations despite the reduced data set size, an adaptive method for adapting the learning process is used for this purpose. In addition, an optimization of the input data was considered by incorporating depth information to improve the classification results. For the execution of the trained models, an embedded heterogeneous system based on a Xilinx FPGA SoC was specially developed for this work. The presented camera system offers integrated hardware units to calculate depth information in real time. For this procedure, own algorithms for depth determination from stereo images were researched and implemented efficiently as a hardware-based solution. The depth information thus provided could be used to further enhance classification by using a segmentation method. This was confirmed in test series. For the developed camera system, existing implementations of binary neural networks were adapted and extended by an own implementation of a coordinated network architecture. Extensive investigations were carried out to find an optimal architecture for the special conditions of automatically generated data sets. The resulting AL BNN Framework can independently train these by specifying desired object classes and finally delivers models with an excellent classification accuracy

of 82.3% for execution on GPU-based systems and the newly developed embedded camera system “AeonCam”. This means that new objects can be trained in less than an hour thanks to the optimized training procedure of the AL BNN Framework, thus flexibly expanding the functionality of the camera system for many object classification applications. The classification could be realized by the hardware support with an image refresh rate of up to 108 images per second with a very low power consumption of less than 5W. As a result, the presented system is an excellent choice for implementing an object classification, especially for use in mobile robots and drones, where energy efficiency plays an important role. Here also the compact construction of the embedded system of 110mm x 75mm x 31mm can convince. This paper shows how automated learning can be applied to heterogeneous systems so that embedded systems can independently create new data sets and thus learn the classification of new objects.

Glossar

ASIC	Application-specific integrated circuit
ADAM	Adaptive Moment Estimation
AlexNet	Netzarchitektur
BNN	Binary Neural Network
CIFAR10	Bilddatensatz zum Trainieren von Neuronalen Netzen
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DDR	Double Data Rate Memory
DNN	Deep Neural Network
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High Level Synthese
ImageNet	Bilddatensatz zum Trainieren von Neuronalen Netzen
LUT	Look-up Table (Logikelemente eines FPGAs)
MNIST	Datensatz zum Trainieren von Handschrifterkennung
PM	Personenmonat, Angabe des Arbeitszeitaufwands
RAM	Random Access Memory
SGD	Stochastic Gradient Descent
SoC	System on Chip
SQNR	Signal to quantization noise ratio
SRCC	Sparse Retina Census Correlation
SVHN	Street View House Numbers Datensatz

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Einsatz von Neuronalen Netzen

Neuronale Netze werden bereits in vielen Bereichen der aktuellen Forschung zur Auswertung von großen Datensätzen verwendet. Auch für industrielle Anwendungen ist eine Umsetzung durch Neuronale Netze sehr interessant, da die Verarbeitung von Datensätzen auf viele unterschiedliche Anwendungen trainiert und somit flexibel angepasst werden kann. In einige Suchmaschinen oder Spracherkennungssystemen kommen bereits erfolgreich Neuronale Netze zum Einsatz. Das AlphaGo Team des Unternehmens DeepMind hat mit [SH16] gezeigt, dass eine künstliche Intelligenz in Form eines Computerprogramms einen menschlichen Spieler in der Anwendung des Strategiespiels "GO" besiegen kann. Hierfür wurden die bisherigen Ansätze durch eine Kombination aus einem weiterentwickelten "tree search" Algorithmus und einem Neuronalen Netz erweitert, um den möglichen Spielverlauf vorherzusagen und somit die optimale Entscheidung für den jeweiligen Spielzug treffen zu können. Diese Variante aus dem Bereich des "Reinforcement Learning" hat die großen Potenziale des maschinellen Lernens aufgezeigt.

Die Idee, ein geplantes Verhalten im gewünschten Einsatzbereich durch "einfaches" Antrainieren zu erreichen, scheint leicht umsetzbar und verspricht viele Einsatzmöglichkeiten. In der Realität werden zur Erstellung und Trainieren eines Netzes komplexe Frameworks wie Caffe [JSD+14] , Tensorflow [ABC+16], Theano [ARAA+16] oder Keras [C+15] eingesetzt. Zusätzlich werden große Datenmengen benötigt, um über langwierige Trainingsphasen und Optimierungsparameter passende Gewichte für die gewählte Netzarchitektur zu bestimmen. Dabei beschränkt sich ein Großteil der aktuellen Forschungen auf die Optimierung der Netzarchitektur für verfügbare Trainingsdatensätze (CIFAR-10/100 [Kri17], ImageNet [RDS+15], MNIST [Den12]). In dieser Arbeit soll der Einsatz Neuronaler Netze am Beispiel der Objektklassifikation auf heterogenen Systemen untersucht und optimiert werden. Hierzu werden Implementierungen auf verschiedenen Systemen umgesetzt

und hinsichtlich ihrer Performanz und Energieeffizienz verglichen. Gleichzeitig sollen die Herausforderungen bei einer Umsetzung in realen Szenarien aufgezeigt und durch entsprechende Aufbereitung der Eingangs/Trainingsdaten verbessert werden. Hierzu soll im Zuge dieser Arbeit ein Framework zur automatisierten Erstellung und Trainieren eigener Trainingsdaten umgesetzt werden.

1.2 Heterogene Systeme zur Beschleunigung von Neuronalen Netzen

Es werden folgende Schlüsselelemente zur Umsetzung Neuronaler Netze auf heterogenen Systemen thematisiert:

- Beschleunigung des Trainings und der Ausführung auf heterogenen Systemen
- Verbesserung der Eingangsdaten
- Reduzierung des Datensatzes
- Automatisierte Generierung von Trainingsdaten
- Optimierung der Netzarchitektur

In den Kapiteln Grundlagen (2) und Lernen von Neuronalen Netzen (3) sollen zunächst die Grundbegriffe im Bereich der Neuronaler Netze, sowie die generellen Komponenten zur Umsetzung einer Objektklassifikation mittels Neuronaler Netze und die Besonderheiten und Vorzüge von heterogenen Systemen vorgestellt werden. Das nachfolgende Kapitel (4) befasst sich mit verwandten Arbeiten zur Umsetzung Neuronaler Netze auf heterogenen und FPGA-basierten Systemen. Im Anschluss wird die Zielplattform für die geplante Umsetzung in Kapitel (5) definiert und die Idee des Automatisierten Lernens in Kapitel (6) vorgestellt. Das Kapitel (6.3) zur Evaluation Neuronaler Netze befasst sich mit der Auswahl und Optimierung verschiedener Netzarchitekturen, Auflösungen und Eingangsdaten für die Implementierung der geplanten Objektklassifikation. Nach Auswahl der geeigneten Verfahren zur Umsetzung wird im Kapitel (7) “Hardware Design” die Implementierung des Systems auf der FPGA-basierten Zielplattform beschrieben. Im darauf folgenden Kapitel (8) werden die Ergebnisse dieser Umsetzung einschließlich der Trainings- und Ausführungsergebnisse vorgestellt. Im abschließenden Kapitel (9) wird die Umsetzung des Automatisierten Lernens sowie die Ergebnisse der hardwarebasierten Objektklassifikation zusammengefasst und reflektiert.

1.3 Eigene Beiträge dieser Arbeit

Im Verlauf dieser Arbeit wird das Thema des Automatisierten Lernens intensiv betrachtet und aus den Erfahrungen im Verlauf dieser Arbeit heraus ein eigenes Framework zum Automatisierten Lernen von Neuronalen Netzen entwickelt. Hierfür wird eine umfangreiche Evaluation zur Verbesserung der Eingangsdaten und zur Anpassung der Netzwerkarchitektur sowie der Trainingsparameter durchgeführt. Zur technischen Umsetzung wird in dieser Arbeit ein eigenständiges heterogenes Kamerasystem entwickelt, das über einen Stereo-Kameraaufbau und integrierter Hardware-beschleunigter Stereo-Verarbeitung Tiefen zur Verbesserung der Objektklassifikation zur Verfügung stellt und weitere FPGA-Ressourcen zur Beschleunigung bestehender Implementierungen Binärer Neuronaler Netze nutzt (siehe Bild 1.1).

Abbildung 1.1: Heterogene Stereokamera-Plattform AeonCam

Dabei wurde der elektrische und mechanische Designprozess der Kamerahardware und die Implementierung eigener Bildverarbeitungsalgorithmen als FPGA-basiertes Design durchlaufen, um eine intelligente Lösung in Form einer eingebetteten Kamera mit integrierter Verarbeitung zur Berechnung von Tiefenbildern und der weiteren Auswertung mittels Hardware-beschleunigter Binärer Neuronaler Netze umzusetzen. Die Implementierung Binärer Neuronaler Netze wurde für das neu entwickelte System-Design optimiert und getestet. Diese Arbeit zeigt die Vorteile des Automatisierten Lernens durch das neu entwickelte AL BNN Framework zur automatisierten Generierung von Datensätzen als Anwendung Neuronaler Netze zur Objektklassifikation durch Binäre Neuronale Netze. Besonders durch die Abstimmung des Trainingsprozesses auf automatisch generierte Datensätze unter Anwendung bestehender Verfahren zur adaptiven Anpassung des Lernprozesses konnten die Trainingsergebnisse deutlich verbessert werden.

Kapitel 2

Grundlagen

Dieses Kapitel beschreibt die benötigten Basiselemente für die Umsetzung eines Neuronalen Netzes auf heterogenen Systemen. Es wird der grundlegende Aufbau Neuronaler Netze, sowie spezielle Optimierungen zur Beschleunigung von Neuronalen Netzen und ein Vergleich der Performanz auf heterogenen Systemen vorgestellt.

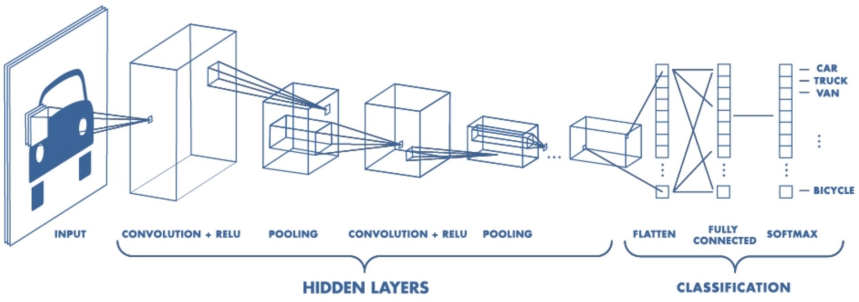

Abbildung 2.1: Übersicht des generellen Lagenaufbaus eines Convolutional Neural Network, Quelle: [TM17]

2.1 Aufbau von Neuronalen Netzen

Die Grundlagen zum Aufbau und der Funktionsweise von Neuronalen Netzen wurden aus den Quellen [TM17], [Kar18] entnommen und entsprechend für diese Arbeit aufbereitet und zusammengefasst. Neuronale Netze bestehen in der Anwendung des "Deep Learning" aus einem Aufbau aus mehreren Lagen (Deep Neural Network, DNN). Eine spezielle Form der Neuronalen Netze ist das sogenannte "Convolutional Neural Network" (CNN), das häufig für Bildverarbeitungsaufgaben herangezogen wird. Grundsätzlich bestehen sie, wie klassische Neuronale Netze, aus einer Vielzahl von Neuronen, die trai-

nierbare Gewichte und Bias-Werte besitzen. Jedes Neuron bekommt einen Eingangswert zugewiesen und führt entsprechend eine Punkt-Multiplikation gefolgt von einer optionalen nicht-linearen Operation durch. Im Gegensatz zu den konventionellen Neuronalen Netzen werden die Eingangsdaten allerdings nun als Bildinformationen angenommen. Hier werden die Eingangsdaten als 2D Matrizen bestehend aus Pixelwerten repräsentiert und folgen somit einer festen 2-dimensionalen Struktur. Dieses Netz arbeitet mit Filteroperationen mit einer festen Kerngröße in einem mehrstufigen Aufbau, die entsprechend der Architektur des Netzes angelegt werden. Die Basiselemente sind hier "Convolutional", "Normalization", "Pooling" und "Activation" Lagen. Bild 2.1 zeigt den generellen Aufbau dieser Lagen.

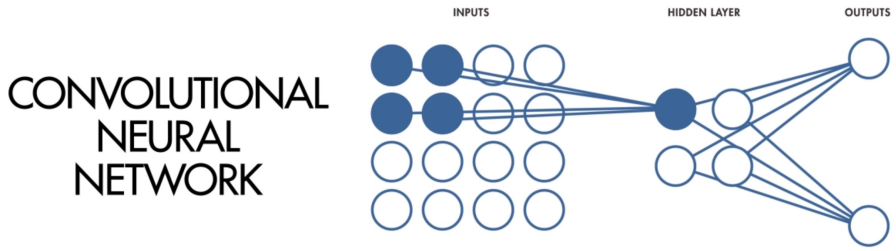

Abbildung 2.2: Aufbau eines CNNs mit Hidden Layer Struktur, Quelle: [TM17]

Ein Neuronales Netz, wie in der Darstellung 2.2 gezeigt, soll eine Aussage über einen Bildausschnitt treffen. Hierfür werden die Daten des Input Layers über Filteroperationen zu mehreren sogenannten "Hidden Layer" verbunden. Dabei ist eine Besonderheit eines Convolutional Neural Networks, dass nicht alle Eingänge mit einem Neuron aus dem Hidden Layer Segment verbunden sind (siehe Bild 2.2). Dies entspricht der Wahl eines lokalen rezeptiven Feldes (local receptive field), das jeweils an ein Neuron der Hidden Layer angeschlossen ist.

2.1.1 Input Layer

Der Eingang eines CNNs entspricht einem gewählten Bildausschnitt und besteht aus Pixeln. Dabei sind die Pixelwerte von farbigen Eingangsbildern in 3 Lagen (Rot, Grün, Blau) aufgeteilt. Dies entspricht bei einer Eingangsauflösung von 32x32 Pixeln einem Dateneingang von 32x32x3.

2.1.2 Convolutional Layer

Der Convolutional Layer führt Filteroperationen in Form einer Bildfaltung mit einer festen Kerngröße (z.B. 2x2), wie in der Darstellung in 2.2) gezeigt,

aus. Die Filteroperation entspricht einer elementweisen Multiplikation mit den im Training bestimmten Gewichten in Matrixrepräsentation. Zusätzlich wird ein Bias hinzugefügt. Die unten aufgeführte Formel 7.14 zeigt die Berechnung eines Convolutional Layers mit M Eingängen in Form von sogenannten "feature maps" $x_1, .., x_M$, N Ausgängen in Form von "feature maps" $y_1, .., y_N$ und einer Aktivierungsfunktion f.

$$y_n = f(\sum_{m=1}^{M} x_m * w_{n,m} + b_n) \tag{2.1}$$

Die Parameter umfassen entsprechend $MxNxKxK$ Gewichte und N Bias-Werte.
Um die Details der Berechnung zu verdeutlichen, ist in Bild 2.3 eine Faltung eines RGB Eingangsbildes der Größe 5x5 mittels zwei Filterkernen (K=2) der Dimension 3x3 dargestellt.

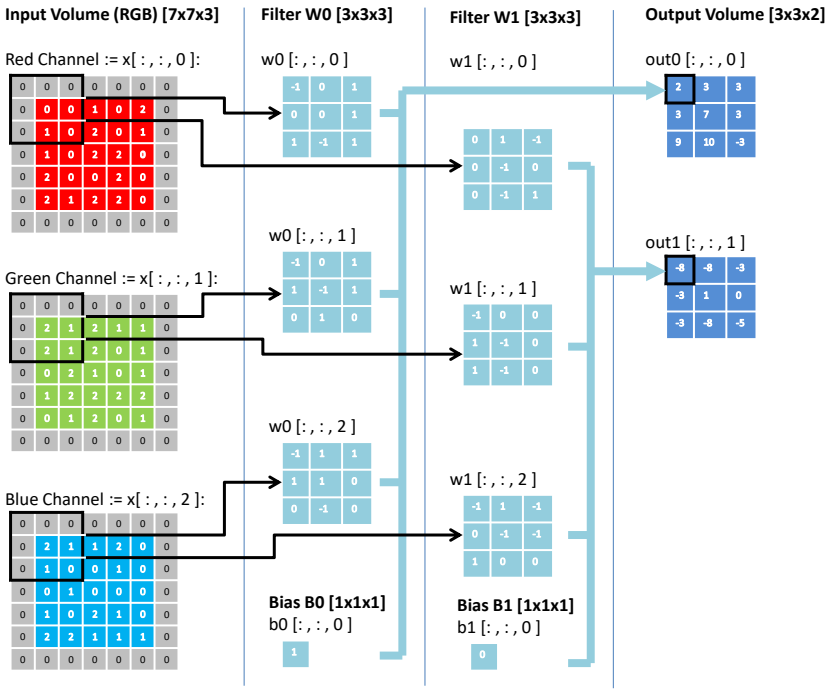

Abbildung 2.3: Beispielberechnung einer Faltung mit 5x5 Eingangsbild (Schritt 1)

In diesem Beispiel wird zusätzlich die übliche Methode des "Padding" angewendet (Zero-Padding = 1), welche entsprechend der angegebenen Pixelzahl einen Rahmen aus Nullen um die Eingangsdaten legt. Durch dieses Verfahren können bei Filteroperationen die Eingangsdimensionen erhalten werden.

Um den ersten Wert der Ausgangsmatrix out0 zu bestimmen, wird die markierte Bildregion aus x für jeden Kanal $[0, 1, 2]$ elementweise an die Gewichte $W0[0, 1, 2]$ multipliziert. Zur Bestimmung des nächsten Elements wird das Fenster, wie in Bild 2.4 dargestellt, verschoben. Dabei wird die Verschiebung als "Stride"- Wert (Stride =2) angegeben. Durch die angewendete Unterabtastung reduziert sich die Dimension der Ausgangsmatrizen out0, out1 auf $3x3$. Die Anzahl der Ausgangsmatrizen ist durch die Anzahl der angewendeten Filterkerne (K=2) bestimmt. So ergibt sich in diesem Beispiel ein Ausgangsvolumen der Größe $3x3x2$. Es wird zusätzlich ein Bias hinzugefügt. Dadurch ergibt sich folgenden Gleichung:

$$out_0 = x * w_0 + b_0 \qquad out_1 = x * w_1 + b_1 \tag{2.2}$$

Die eingeführten Gewichte (W0,W1) und die Bias-Werte (B0,B1) werden, wie zuvor erwähnt, durch ein Trainingsverfahren bestimmt.

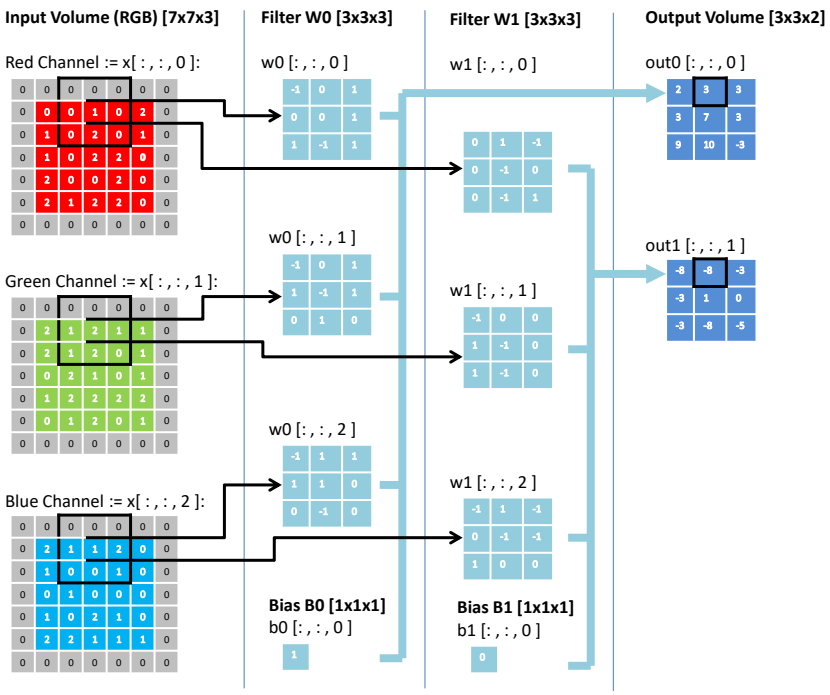

Abbildung 2.4: Beispielberechnung einer Faltung mit 5x5 Eingangsbild (Schritt 2)

2.1.3 Activation Layer

Diese Lage wendet eine Transformation über eine gewählte Aktivierungsfunktion auf jedes Element der jeweiligen Lage an. Ein einfaches Beispiel

ist hier das ReLu Element (Rectified Linear Unit), das entsprechend der Aktivierungsfunktion

$$\phi(x) = \max(0, x) \tag{2.3}$$

beschrieben wird. Die Ausgänge eines Neurons werden demnach dem höchsten positiven Wert zuordnet. Dabei werden negative Werte zu Null gesetzt. Bild 2.5 zeigt die Aktivierungsfunktion des ReLu Elements.

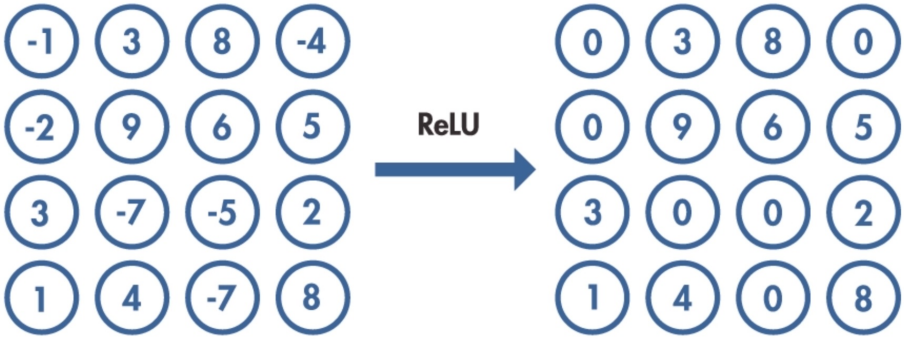

Abbildung 2.5: Aktivierungsfunktion des ReLu-Elements, Quelle: [TM17]

Neben der ReLu Aktivierungsfunktion können auch folgende Aktivierungsfunktionen angewendet:

$$\text{Sigmoid Aktivierungsfunktion} = \quad \phi(x) = \frac{1}{1 + \mathrm{e}^{-x}} \tag{2.4}$$

$$\text{Softmax Aktivierungsfunktion} = \quad \phi(x)_j = \frac{\mathrm{e}^{x_j}}{\sum_{k=1}^{K} \mathrm{e}^{x_j}} * \tag{2.5}$$

$$\text{Tanh Aktivierungsfunktion} = \quad \phi(x) = \tanh(x) \tag{2.6}$$

$$\text{Softplus Aktivierungsfunktion} = \quad \phi(x) = \log(1 + \mathrm{e}^{x}) \tag{2.7}$$

$$\text{Exponential Lin. Unit} = \quad \phi(x) = (x > 0)?x : e^{x} \tag{2.8}$$

$$\text{Lineare Aktivierungsfunktion} = \quad \phi(x) = x \tag{2.9}$$

(* wobei K die Zahl der Neuronen darstellt)
Die Auswahl einer geeigneten Aktivierungsfunktion ist eine Designentscheidung, die in Abhängigkeit von der Netzarchitektur und des Trainingsdatensatzes getroffen werden muss. Häufig wird am Ende der Netzarchitektur die Softmax Funktion verwendet, um als Ausgang eine Aussage in Form einer Wahrscheinlichkeit zu bekommen.

2.1.4 Pooling Layer

Der Pooling Layer reduziert die Daten einer Lage durch Zusammenführen benachbarter Regionen. Im Fall des $2x2$ Pooling-Verfahrens werden die höchsten Elemente einer $2x2$ Region als neuer Datenwert angenommen. Die Größe des Datenarrays reduziert sich in diesem Fall in zwei Dimensionen auf die Hälfte ($4x4$ auf $2x2$). Dabei bleibt bei 3-dimensionalen Strukturen (Voluminа), bestimmt durch die Anzahl der verwendeten Kerne, entsprechend die dritte Dimension unverändert. Übliche Pooling Größen sind $2x2$ oder $3x3$.

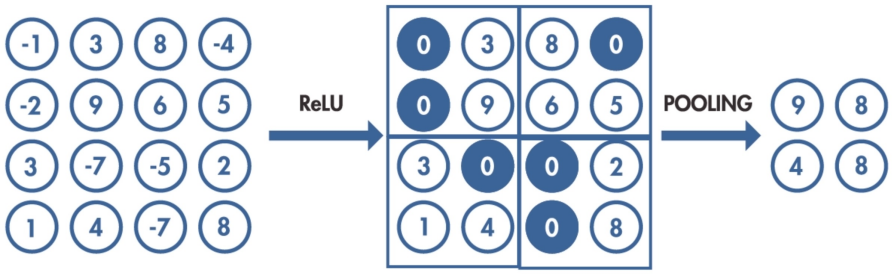

Abbildung 2.6: Pooling-Verfahren am Beispiel des ReLU Elements, Quelle: [TM17]

Diese Lage wird periodisch zwischen den Convolutional Lagen eingefügt, um die Parameter in einer Netzarchitektur zu komprimieren und gleichzeitig "overfitting" zu verhindern. Der Begriff "overfitting" beschreibt hier ein Phänomen während des Trainingsprozesses, dass sich bei übermäßigem Training eine Übersättigung der Trainingsgewichte einstellen kann. Dies wirkt sich negativ auf den Trainingsverlauf durch rückläufige Trainingsergebnisse aus und sollte demnach vermieden werden.

2.1.5 Dense Layer (Fully Connected)

Der Dense-Layer - auch Fully Connected Layer genannt - bekommt einen Eingangsvektor bestehend aus $1x1$ Elementen (Pixeln) und führt ein Punkt-Produkt mit einem Gewichtsvektor der gleichen Größe durch. Das Ergebnis des Punkt-Produkts wird zusammen mit einem Bias-Wert aufsummiert und über eine Aktivierungsfunktion ausgewertet. Die Formel 2.10 zeigt die Berechnung eines Dense Layers mit M Eingangswerten, N Ausgangswerten und der Aktivierungsfunktion f.

$$y_n = f(\sum_{m=1}^{M} x_m \cdot w_{n,m} + b_n) \tag{2.10}$$

Kapitel 3

Lernen von Neuronalen Netzen

Nachdem im vorherigen Kapitel die Lagen eines Neuronalen Netzes dargestellt wurden, soll in diesem Abschnitt der allgemeine Ablauf des Trainingsprozesses Neuronaler Netze erläutert werden, da die Optimierung des Lernprozesses eines der Kernelemente dieser Arbeit darstellt. Bei der Anwendung Neuronaler Netze muss zwischen zwei essenziellen Phasen, wie in Abbildung 3.1 dargestellt, unterschieden werden.

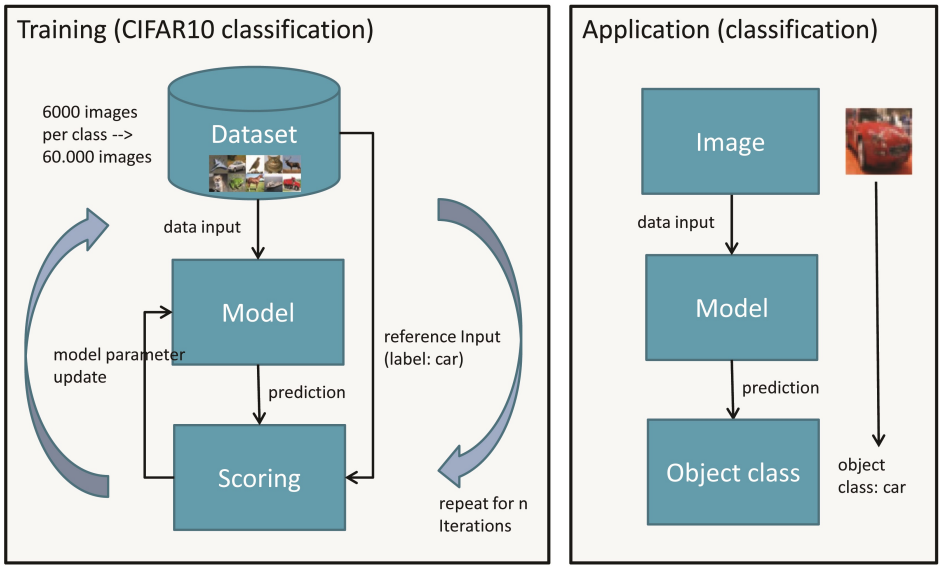

Abbildung 3.1: Training und Ausführung von Neuronalen Netzen am Beispiel des CIFAR-10 Datensatzes [Kri17]

Dies soll anhand des Beispiels einer Objektklassifikation des CIFAR-10 Datensatzes genauer beschrieben werden. Der CIFAR-10 Datensatz besteht aus 10 verschiedenen Objektklassen aus den Bereichen Fortbewegungsmittel und Tiere. Die Klassen des Datensatzes sind in Grafik 3.2 veranschaulicht. Dabei sind die Klassen “car” und “truck” eigenständige Klassen und somit vollständig voneinander getrennt. Zusammen mit dem Bildmaterial sind Referenzen für jedes Bild im Datensatz gespeichert. Für jedes Bild ist somit ein Referenzlabel in Form von Zahlenwerten [0..9] verfügbar und entsprechend der gezeigten Klasse zugeordnet. Bild 3.1 zeigt den schematischen Ablauf der Trainingsphase in Gegenüberstellung zur Ausführungsphase.

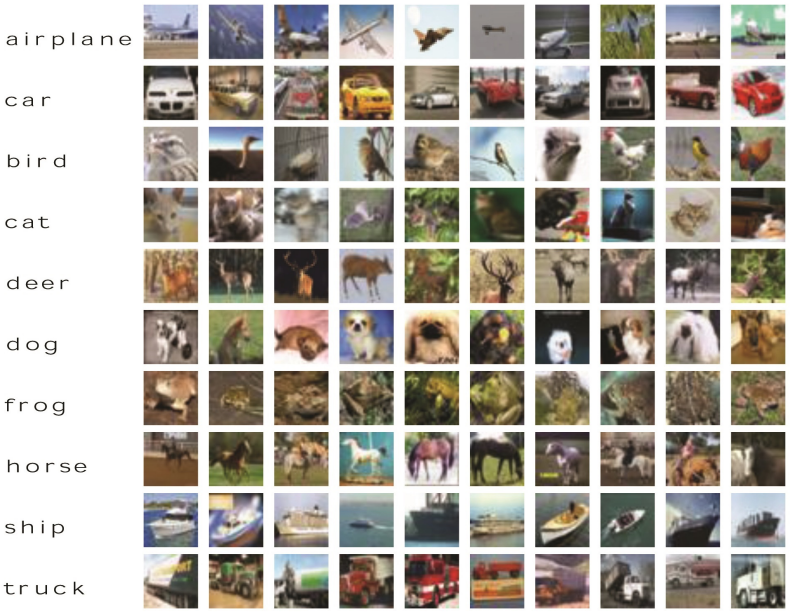

Abbildung 3.2: Klassen des CIFAR-10 Datensatzes, Quelle: [Kri17]

Während der Trainingsphase werden die Bilder aus dem Trainingsdatensatz in kleinere Mengen (Batches) unterteilt und separat als Dateneingang an das Modell angelegt. Dabei umfasst das Model sowohl die Netzarchitektur als auch die trainierbaren Parameter (Gewichte, Aktivierungselemente, Bias-Werte). Über die Berechnungen jeder Lage, die in Kapitel 2.1 vorgestellt wurden, wird eine Aussage über den Bildinhalt in Form eines Labels der Klasse (im CIFAR-10 Beispiel: [0..9]) getroffen. Diese Berechnung entspricht dem Vorwärtspfad durch die Netzstruktur.
Die Aussage über das Klassenlabel wird anschließend mit dem Referenzlabel aus dem Datensatz verglichen. Dadurch kann entsprechend der Berechnung des Rückwärtspfades eine Anpassung der trainierbaren Modellparame-

ter vorgenommen werden. Dabei kann die Geschwindigkeit der Anpassung über die Lernrate konfiguriert werden. Dieser Ablauf wird für alle Bilder des Datensatzes wiederholt. Der vollständige Durchlauf eines Datensatzes wird als Epoche bezeichnet. Die komplette Trainingsphase kann hierbei, abhängig vom Datensatz, ca. 50 bis 1000 Epochen betragen. Zur Ausführungsphase werden lediglich die Berechnungen des Vorwärtspfads benötigt. Hier werden zur Klassifikation einzelne Bilder an das Modell angelegt. Die vortrainierten Gewichte sind hierfür fest in das Modell geladen. Entsprechend wird nun eine Aussage für die Klasse des Eingangsbildes getroffen. Im Diagramm 3.1 zeigt das Eingangsbild aus dem CIFAR-10 Datensatz ein Auto und wird bei erfolgreichem Training der Klasse “car” zugeordnet.

3.1 Optimierung von Neuronalen Netzen

3.1.1 Quantisierung

Zur Optimierung von Neuronalen Netzen ist die Quantisierung eine Methode, um die Modellgröße eines Neuronalen Netzes drastisch zu reduzieren. Durch diese Reduktion können die Ausführung und das Training eines Netzes beschleunigt werden. Bei diesem Ansatz werden die internen 32-Bit Gleitkomma Berechnungen und die zugehörigen Datenmengen (Gewichte, Aktivierungselemente) zu Festkommazahl-Berechnungen umgewandelt. Die Arbeit [LTA16] hat sich mit der Untersuchung von Quantisierung Neuronaler Netze in verschiedenen Bitbreiten befasst. Dabei werden zwei generelle Herangehensweisen an eine Quantisierung vorgestellt:

1. Konvertierung eines im Vorfeld trainierten 32-Bit Gleitkomma-Modells zu einem Festkommazahl-Modell ohne erneutem Training

2. Aufbau eines neuen Trainingsmodells und einer Netzwerkarchitektur mit Festkommazahl-Restriktionen

Die vorgestellte Arbeit sieht hier deutliche Vorteile für die erreichbare Präzision durch Trainingsanpassungen, fokussiert allerdings zur Untersuchung auf die erste Methode. Für die Optimierung vortrainierter Netze wird eine Optimierungsstrategie basierend auf dem Verhältnis zwischen Signal und Quantisierungsrauschen (signal-to-quantization-noise-ratio: SQNR) eingeführt. Dabei wird zunächst die Auswahl von Bitbreiten für verschiedene Eingangsverteilungen untersucht.

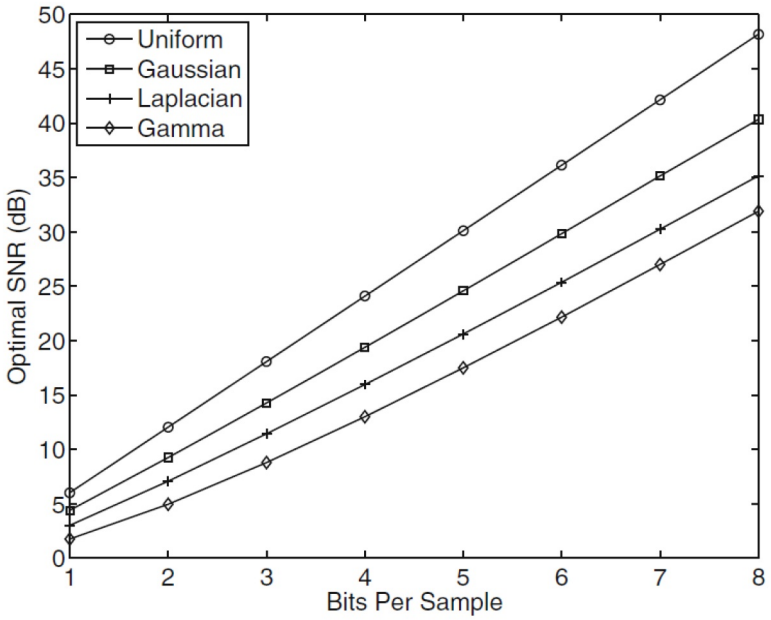

Abbildung 3.3: Optimale SQNR durch universelle Quantisierung von universeller, Gaussian, Laplacian und Gamma Verteilungen, Quelle: [LTA16]

Bild 3.3 zeigt die SQNR als Funktion der Bitbreite unter Annahme optimaler Größenumwandlung und idealer Eingangsdaten. Das Bild zeigt eine annähernd lineare Abhängigkeit zwischen der SQNR und der Bitbreite. Es gilt

$$\gamma_{dB} = \kappa \cdot \beta \tag{3.1}$$

wobei $\gamma_{dB} = 10 \log_{10}(\gamma)$ die SQNR in dB formuliert und κ die Quantisierungs-Effizienz darstellt. β entspricht hierbei der Bitbreite. Für die Anwendung in Neuronalen Netzen haben die Autoren sowohl für die Verteilungen der Gewichte, als auch für die Verteilung der Aktivierungen basierend auf ihren Untersuchungen eine Gauß-Verteilung angenommen. Weiterhin wird angenommen, dass ein größerer SQNR-Wert zu einer ungenaueren Klassifikation führt. Durch diese Annahmen konnte ein Modell zur Minimierung der SQNR für jede Lage und damit auch die benötigte Bitbreite für die Gesamtarchitektur der CIFAR-10 [Kri17] und AlexNet [KSH12] Implementierung bestimmt werden. Für die CIFAR-10 Architektur kann nach den Berechnungen der Arbeit [LTA16] die Bitbreite der letzen Lage um 8 Bit im Bezug auf die Eingangsbitbreite reduziert werden, ohne die Genauigkeit maßgeblich zu beeinflussen. Durch äußere Restriktionen soll aber häufig eine gemeinsame

Bitbreite für das Gesamtnetz gewählt werden, deshalb sind die realen Messungen der Fehlerraten von quantisierten Architekturen mit den ausgewählten Bitbreiten besonders interessant. Diese Ergebnisse sind in Tabelle 3.1 aufgeführt.

Aktivierungs-	Bit-Breite der Gewichte			
Bit-Breite	4	8	16	float
4	8,30	7,50	7,40	7,44
8	7,58	6,95	6,95	6,78
16	7,58	6,82	6,92	6,83
float	7,62	6,94	6,96	6,98

Tabelle 3.1: Fehlerraten von quantisierten Architekturen mit verschiedenen Bitbreiten, Quelle: [LTA16]

Die CIFAR-10 Implementierungen mit Aktivierungselementen und Gewichten in Gleitkomma-Präzision ist als Referenz mit einer Fehlerrate von 6,98% angegeben. Die 8-Bit Quantisierung hat hierbei mit einer Fehlerrate 6,95% sogar bessere Ergebnisse geliefert. Selbst die Variante mit einer 4-Bit Quantisierung weist noch eine gute, wenn auch etwas höhere, Fehlerrate auf.

3.1.2 Binäre Neuronale Netze

Die Arbeit [CHS+16] zur Umsetzung von Binären Neuronalen Netzen stellt eine Implementierung eines Neuronalen Netzes mit 1-Bit Präzision vor. Im Gegensatz zur vorherigen Arbeit wurde hier die Architektur vollständig auf die binäre Repräsentation von Aktivierungen und Gewichten ausgelegt und das vorgestellte Binäre Neuronale Netz mit diesen Restriktionen neu trainiert. Dabei wurde für Gewichte und Aktivierungen die gleiche Restriktion in Form von

$$x^b = Sign(x) = \begin{cases} +1 & \text{für } x \geq 0 \\ -1 & \text{andernfalls} \end{cases} \tag{3.2}$$

angewendet. In dieser Gleichung steht x^b für die binarisierte Darstellung des realen Werts x. Zusätzlich wurde eine zweite stochastische Binarisierungsfunktion experimentell angewendet:

$$x^b = \begin{cases} +1 & \text{mit Wahrscheinlichkeit } p = \sigma(x) \\ -1 & \text{mit Wahrscheinlichkeit } 1 - p \end{cases} \tag{3.3}$$

Hierbei ist σ die "hard sigmoid" Funktion nach

$$\sigma(x) = clip(\frac{x+1}{2}, 0, 1) = max(0, min(1, \frac{x+1}{2})) \tag{3.4}$$

Durch die komplexe Berechnung der stochastischen Funktion, besonders durch die benötigten Hardwarefunktionen zur Generierung von zufälligen Bits, wurde in Verlauf der Arbeit [CHS+16] ausschließlich die deterministische Funktion 3.2 angewendet. Tabelle 3.2 zeigt die resultierenden Fehlerraten der CIFAR-10 und der SVHN Klassifizierungen im Vergleich mit der binarisierten Architektur.

Binarized activations+weights, during training and test		
Data set	SVHN	CIFAR-10
BNN (Torch7)	2,53%	10,15%
BNN (Theano)	2,80%	11,40%
No binarization (standard results)		
Data set	SVHN	CIFAR-10
Maxout Networks [GWFM+13]	2,47%	11,68%
Network in Network [LCY13]	2,35%	10,41%
Gated pooling [LGT16]	1,69%	7,62%

Tabelle 3.2: Vergleich der Fehlerraten der CIFAR-10 und SVHN Klassifikationen, Quelle: [CHS+16]

Es wird deutlich, dass die Fehlerraten des Binären Neuronalen Netzes mit den State-of-the-Art Implementierungen [GWFM+13],[LCY13] mithalten können. Je nach gewähltem Framework sind die Ergebnisse sogar besser als ohne binäre Quantisierung. In Bild 3.4 sind die Trainingsverläufe für eine 32-Bit Gleitkomma-"Baseline"-Implementierung und der BNN Varianten im Vergleich dargestellt.

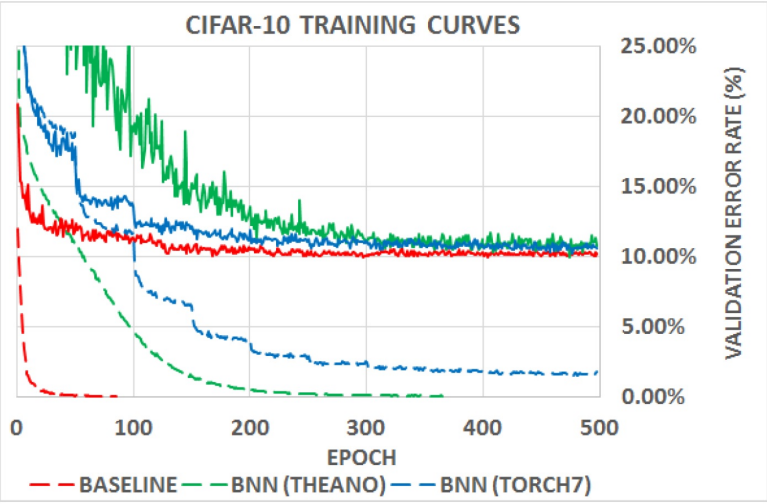

Abbildung 3.4: Trainingskurven von binären und 32-Bit Gleitkomma CNN Implementierungen, Quelle: [CHS+16]

Es zeigt sich, dass BNNs zwar langsamer zu trainieren sind, aber annähernd eine vergleichbare Präzision erreichen können. Besonders interessant sind allerdings die Performanzmessungen der Veröffentlichung [CHS+16]. Es konnte durch einfache Optimierung der GPU Kernel (XNOR) die Ausführung der MNIST [Den12] (Datensatz zur Handschrifterkennung) Klassifikation um den Faktor 7 beschleunigt werden.

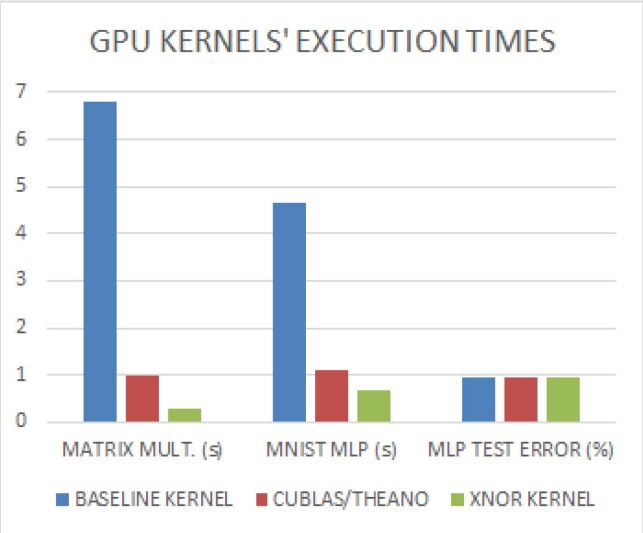

Abbildung 3.5: Performanzvergleich zwischen Baseline und binärer GPU-Kernel-Implementierungen, Quelle: [CHS+16]

Diese Werte sind zusammen mit der Performanz einer "Baseline" und einer CuBlas [Nvi08] Implementierung für die MNIST MLP und Matrix-Multiplikation in Grafik 3.5 dargestellt. Die binäre Matrix-Multiplikation konnte sogar um den Faktor 23 gegenüber der "Baseline"-Implementierung beschleunigt werden.

3.1.3 Batch-Größenoptimierung

Das Training von Modellen Neuronaler Netze wird über ein Gradientenabstiegsverfahren [Bro17] realisiert. Das Gradientenabstiegsverfahren (Gradient Descent) ist ein weitverbreitetes Verfahren im Feld des maschinellen Lernens, um die Koeffizienten und Gewichte eines Modells zu bestimmen. Hierbei wird ein iteratives Verfahren angewendet, um die "objectiveness function" nach

$$Q(\omega) = \frac{1}{n} \sum_{i=1}^{n} Q_i(\omega), \tag{3.5}$$

zu minimieren. Dabei ist der Parameter ω, der die Funktion $Q(\omega)$ minimiert, zu bestimmen. Q_i entspricht hier der Beobachtung an der Stelle i des Datensatzes.
Generell kann zwischen drei Verfahren zur Optimierung des Trainingsprozesses und der Modellanpassung während des Trainingsprozesses unterschieden werden:

- Stochastic Gradient Descent
- Batch Gradient Descent
- Mini-Batch Gradient Descent

Diese werden im folgenden Abschnitt genauer erläutert.

Stochastic Gradient Descent

Das stochastische Gradientenabstiegsverfahren (Stochastic Gradient Descent) oder kurz SGD, ist ein weitverbreitetes Verfahren als Abwandlung des normalen Gradientenabstiegsverfahrens. Bei diesem Verfahren wird nach jedem Trainings-Sample ein Fehler zur Anpassung des Modells berechnet. Dabei konvergiert das SGD Verfahren nach:

$$O(\frac{1}{\sqrt{T}})) \tag{3.6}$$

mit T als Angabe der Iterationen.

Batch Gradient Descent

Das Verfahren des "Batch Gradient Descent" ist eine Variation des Gradientenabstiegsverfahrens, das den Fehler nach jedem Trainings-Sample neu bestimmt, aber die Modellanpassung erst nach einem vollen Durchlauf des Datensatzes vornimmt. So werden die Modellanpassungen nach jeder Epoche durchgeführt.

Mini-Batch Gradient Descent

Das Verfahren des "Mini-Batch Gradient Descent" [LZCS14] ist eine Variation des Gradientenabstiegsverfahrens, das den Trainingsdatensatz in kleinere Untermengen (Batches) unterteilt. Nachdem ein "Batch" bearbeitet wurde, wird der Modellfehler berechnet und die Modellparameter entsprechend angepasst. Zusätzlich kann auch eine Summe der Gradienten oder ein Mittelwert der Gradienten über die Batches gebildet werden, um die Varianz der Gradienten zu reduzieren.
Das "Mini-Batch Gradient Descent" Verfahren stellt einen Kompromiss zwischen der Zuverlässigkeit des SGD Verfahrens und der Effizienz des "Batch

Gradient Descent" Verfahrens dar. Dabei konvergiert das "Mini-Batch Gradient Descent" Verfahren mit den Optimierungen aus [LZCS14] nach:

$$O(\frac{1}{\sqrt{bT}})) \tag{3.7}$$

mit T als Angabe der Iterationen und b als Batch-Größe. Durch die Verwendung des "Mini-Batch Gradient Descent" Verfahrens kann durch den Parameter der Batch-Größe die Trainingsgeschwindigkeit optimal auf die Datensatzgröße angepasst werden. Zusätzlich stehen die Informationen zum aktuellen Trainingsverlauf nach jedem Durchlauf eines "Batches" zur Verfügung. Durch die Anwendung einer reduzierten Batch-Größe kann somit der Trainingsprozess deutlich beschleunigt werden.

3.1.4 Verfahren zur adaptiven Anpassung der Lernrate

Nach [Rud18] haben die vorher beschriebenen Verfahren häufig Probleme bei der Konvergenz. Deshalb müssen folgende Herausforderungen zur Optimierung dieser Verfahren adressiert werden:

- Auswahl einer passenden Lernrate
- Anpassung der Lernrate an den gewählten Datensatz
- Vermeidung von lokalen Minima während der Minimierung der nichtkonvexen Fehlerfunktion beim Trainieren Neuronaler Netze

Ein Beispiel, um diesen Problemen entgegenzuwirken, ist die Einführung von Momenten nach [Qia99]. Eine Darstellung des "Stochastic Gradient Descent" Verfahrens (SGD) mit und ohne Verwendung von Momenten ist in Bild 3.6 gezeigt. Hier wird deutlich, wie die Einführung von Momenten helfen kann, die gewünschte Ausrichtung der SGD Funktion zu erzielen und dabei entstehende Oszillationen zu minimieren.

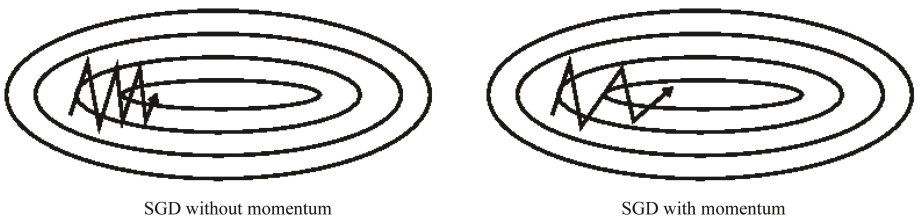

Abbildung 3.6: SGD Verfahren mit und ohne Verwendung von Momenten, Quelle: [Rud18]

Ein im Bereich der Neuronalen Netze sehr verbreitetes Verfahren ist die Berechnung der adaptiven Lernratenanpassung über die "Adaptive Moment Estimation"-Funktion (Adam) aus [KB14].
Hierfür wird zunächst nach der Methode des Gradientenabstiegs der gemittelte letzte Gradient m_t und der quadrierte letzte Gradient v_t nach

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3.8}$$

ermittelt. Daraus können die Momente nach

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.9}$$

bestimmten werden, um anschließend über Anwendung der Adam-Funktion

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.10}$$

die Modellparameter anzupassen.
Weitere Verfahren zur Gradienten-basierten Optimierung der Lernratenanpassung sind:

- Nesterov accelerated gradient [Nes83]
- Adagrad [DHS11]
- Adadelta [Zei12]
- RMSprop [Hin12]
- AdaMax [KB14]
- Nadam [Doz16]
- AMSGrad [RKK18]

Das Adam Verfahren arbeitet jedoch hervorragend in den praktischen Umsetzungen aus [CHS+16] und zeigt in [KB14] deutlich seine Vorzüge gegenüber anderen Verfahren zur adaptiven Anpassung des Lernprozesses. Deshalb wird das Adam Verfahren zur Umsetzung des Trainings in dieser Arbeit ausgewählt.

3.2 Vergleich von Neuronalen Netzen auf heterogenen Systemen

Die Arbeit [NSS+16] vergleicht Implementierungen Neuronaler Netze auf verschiedenen Systemarchitekturen. Es werden hier ebenfalls Binäre Neuronale Netze als optimierte Implementierung von Deep Neural Networks (DNN) eingeführt und die Ausführungszeiten auf CPU, GPU, FPGA und ASIC untersucht. Als Netz-Konfigurationen wurden aktuelle Netze wie AlexNet [KSH12], VGG [SZ14] und NeuralTalk [KFF15] genutzt. Entsprechend dieser Konfigurationen wurde die Performanz relativ zu einer CPU Implementierung und die daraus resultierende Performanz pro Watt bestimmt. Die Ergebnisse dieses Vergleichs sind in Grafik 3.7 und 3.8 dargestellt. Neben den binären Varianten wird auch eine Versuchsreihe mit reduzierter Batch-Größe nach [LZCS14] [BCN18] gezeigt (b10). Diese Optimierungsvariante wurde in Kapitel 3.1 genauer erläutert.

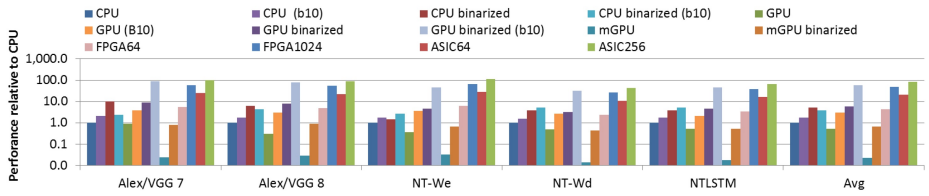

Abbildung 3.7: Performanz von Neuronalen Netzen auf verschiedenen Plattformen in Bezug auf die CPU Basisimplementierung, Quelle: [NSS+16]

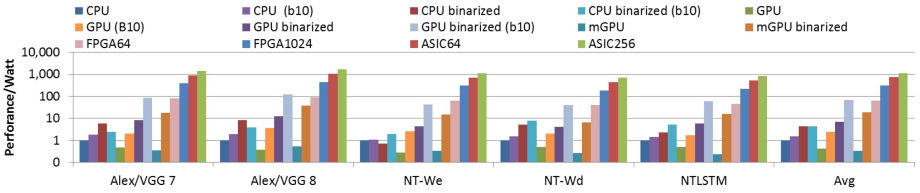

Abbildung 3.8: Performanz pro Watt von Neuronalen Netzen auf verschiedenen Plattformen in Bezug auf die CPU Basisimplementierung, Quelle: [NSS+16]

Die Darstellung zeigt, dass die CPU Implementierung mit einem Leistungsstarken Intel Xeon E5-2699v3 Server-Prozessor beim Vergleich der Performanz deutlich schwächer als die anderen Plattformen abschneidet. Bei der Betrachtung der Performanz pro Watt hingegen liegen sowohl die mobile Nvidia TX1 GPU als auch die leistungsstarke Nvidia Titan X zurück. Die

CPU Implementierung wird in beiden Grafiken als Basis-Referenz mit 1.0 dargestellt. Durch die binäre Implementierung konnte die Performanz im Vergleich zur CPU-Variante mit voller Präzision um den Faktor 5 deutlich beschleunigt werden. Auch durch die Reduzierung der Batch-Größe konnte die Ausführungszeit um 80% reduziert werden. Im Gesamtüberblick zeigt sich deutlich, dass sich folgende Implementierungen in beiden Testreihen hervorheben:

- Binäre Implementierung auf GPU mit reduzierter Batch-Größe
- FPGA Implementierung mit 1024 Recheneinheiten
- ASIC Implementierung mit 64 Recheneinheiten
- ASIC Implementierung mit 256 Recheneinheiten

Die ASIC Implementierungen haben im Vergleich der Performanz und des Energieverbrauchs leichte Vorteile gegenüber den anderen Systemen. Aufgrund der immensen Kosten bei der Entwicklung und Herstellung eines ASICs (Maskenkosten >1M USD) werden diese Systeme für eine reale Umsetzung in dieser Arbeit ausgeschlossen. Entsprechend bleibt für die Umsetzung eines Neuronalen Netzes auf einem heterogenen System in dieser Arbeit die Entscheidung zwischen einer binären GPU Implementierung und einer FPGA-basierten Lösung. Die GPU-Variante verspricht bessere Performanz bei allerdings höherem Energieverbrauch. Die FPGA Implementierung bietet bessere Performanz pro Watt. Beide Systeme sollen im Verlauf dieser Arbeit realisiert und entsprechend der Einsatzmöglichkeiten betrachtet werden.

Kapitel 4

Verwandte Arbeiten

Das vorhergehende Kapitel hat die Vorteile in Bezug auf Energieeffizienz und Performanz einer FPGA-basierten Lösung für die Ausführung Neuronaler Netze gezeigt. Auch für diese Arbeit ist die Umsetzung Neuronaler Netze auf dedizierter Hardware daher eine sinnvolle Schlußfolgerung. Deshalb werden im folgenden Abschnitt einige Implementierungen zur Ausführung und Training von Neuronalen Netzen auf FPGAs vorgestellt.

4.1 ZynqNet: An FPGA-Accelerated Embedded Convolutional Neural Network

In [Gsc16] wird das Potential von FPGA-basierten System-on-Chip Lösungen zur Umsetzung von CNN Implementierungen untersucht. Nach Evaluierung einer geeigneten Netz-Topologie wurde die SqueezeNet Architektur ausgewählt. Es wurden eine Optimierung der Lagenkonfiguration für die Hardwareumsetzung durch Anpassungen hin zur Zweierpotenzen-kompatiblen ZynqNet Variante entsprechend Abbildung 4.1 vorgenommen. Dabei wurde der ARM Prozessor des Zynq SoC zum Laden von Layer-Konfiguration, der Eingangsdaten in Form von vorverarbeiteten Bildern und der trainierten Gewichte in Software genutzt. Außerdem wurde die Softmax-Lage ebenfalls in Software umgesetzt. Die Datenübertragung zum FPGA Design wurde über den AXI-Bus realisiert.
Das Hardwaredesign basiert auf einzelnen Hardwareblöcken zur Parallelisierung nach Bild 4.2. Es arbeiten mehrere 3x3 Berechnungseinheiten (PE) in Form von “Multiply-Accumulate”-Blöcken (MACC) parallel, um die Bildfaltung für jede Lage zu berechnen.

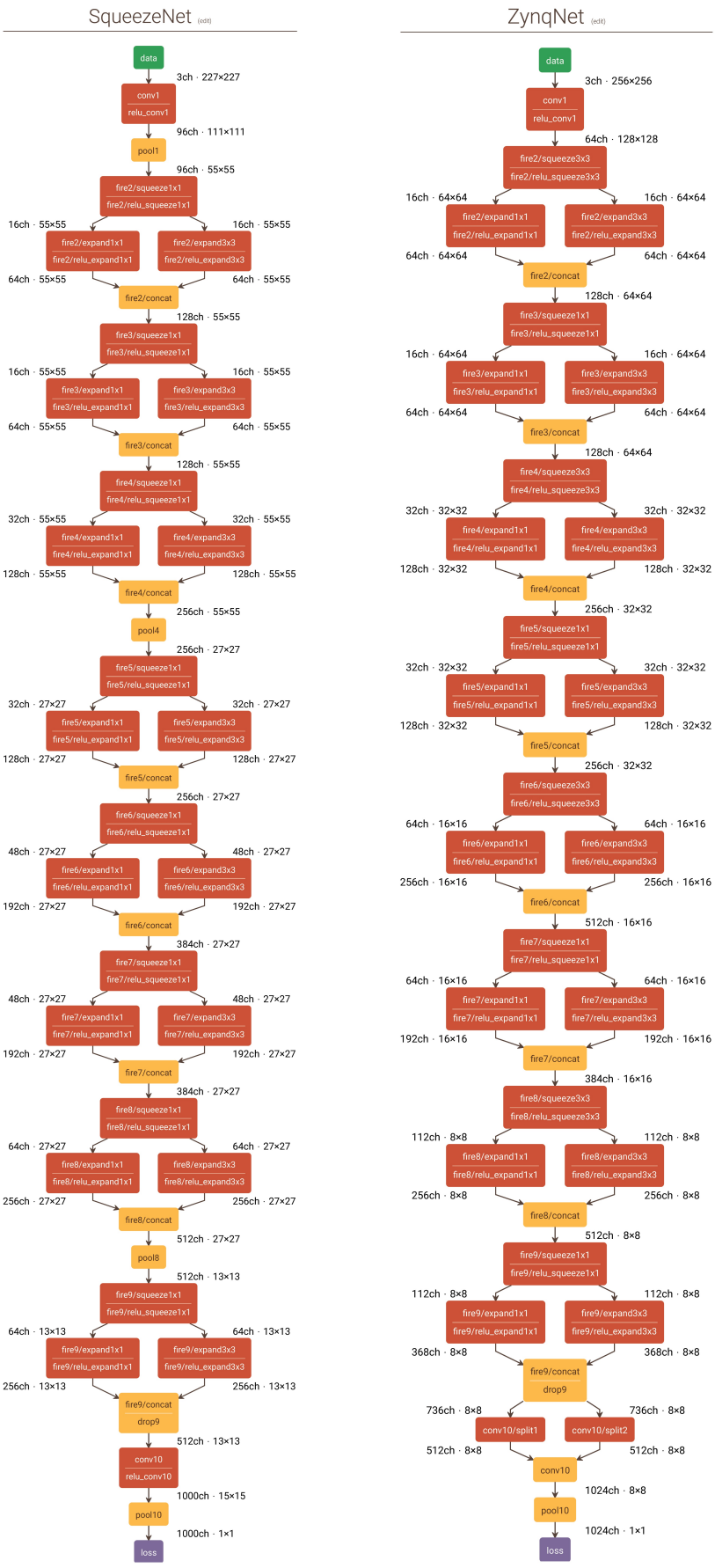

Abbildung 4.1: SqueezeNet und ZynqNet Architektur, Quelle: [Gsc16]

Durch die Hardware-basierte Umsetzung konnte trotz der Anpassungen der Architektur die Fehlerrate der Klassifikation des ImageNet Datensatzes im Vergleich zum ursprünglichen SqueezeNet weiter verbessert werden. So konnte die Fehlerrate vom SqueezeNet (19,7%) und die des AlexNet (19,7%) mit dem ZynqNet (15,4%) übertroffen werden. Lediglich die komplexeren Netze wie VGG-16 (8,1%), GoogLeNet (9,2%) und ResNet-50 (7,0%) schneiden bei der Genauigkeit der Klassifikation besser ab. Hinsichtlich des FPGA Ressourcenverbrauchs lastet die ZynqNet Implementierung, bedingt durch die aufwendige 32-Bit Gleitkomma-Berechnungen, den Zynq XC-7Z045 FPGA fast vollständig aus (BlockRAM: 91%, DSP Blöcke: 82% Logikzellen: 70% [154k LUT]).

Durch die erreichte Verarbeitungszeit von $t_F = 1955\text{ms}$ pro Eingangsbild (0,51 Bilder pro Sekunde) kann die Implementierung nicht überzeugen. Auch bei Betrachtung der Energieeffizienz mit möglicher Optimierung (6,3 Bilder/ Sekunde) nach

$$\eta_{ZynqNet} = \frac{r_F^{'}}{P} = \frac{6.3 frames}{12 Ws} \approx 0.53 \text{ Bilder}/J \tag{4.1}$$

für das ZynqNet System, liegt dieses hinter der Intel Core i7 CPU Variante (1.3 Bilder/J). Die GPU Implementierungen auf Nvidia Titan X und Nvidia Tegra X1 zeigen hier mit nTitanX = 2.5 Bilder/J und nTegraX1 = 8.6 Bilder/J ihre Vorteile.

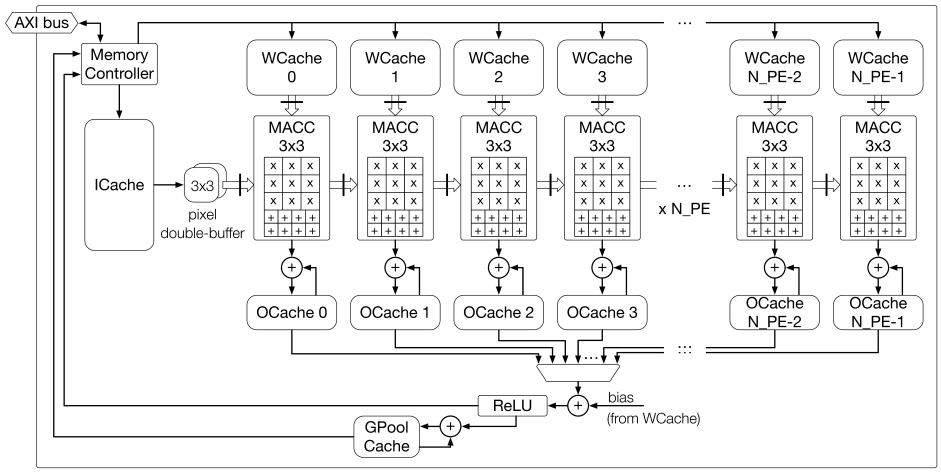

Abbildung 4.2: High-Level Blockdiagramm der FPGA-basierten Beschleuniger des ZynqNet CNNs, Quelle: [Gsc16]

4.2 Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs

In der Arbeit [ZSZ+17] zur Umsetzung Binärer Neuronaler Netze auf FPGAs werden Gewichte und Aktivierungen gemäß der Sign-Funktion aus Kapitel 3.1.2 zu Werten im Bereich von 1 und −1 umgewandelt. Des Weiteren wird die in 2.1 beschriebene Batch-Normalisierungslage eingeführt, um den Informationsverlust während der Binarisierung zu reduzieren. Hierbei werden die Eingangsverteilungen linear verschoben und skaliert, um die Mittelwert- und Einheits-Varianzen zu reduzieren. Dieses Verfahren ist in Gleichung 4.2 gezeigt. Dabei entsprechen x und y dem Eingang und Ausgang der Funktion. μ,σ und ϵ werden statistisch über den Trainingsdatensatz bestimmt, γ und β sind antrainierte Parameter. ϵ wird verwendet, um Rundungsfehler auszugleichen.

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma - \beta \tag{4.2}$$

Die Grundstruktur der binären Netzarchitektur, die in diesem Paper zur Klassifizierung des CIFAR-10 Datensatzes verwendet wurde, ist in Bild 4.3 im Vergleich mit einer regulären CNN Architektur gezeigt.

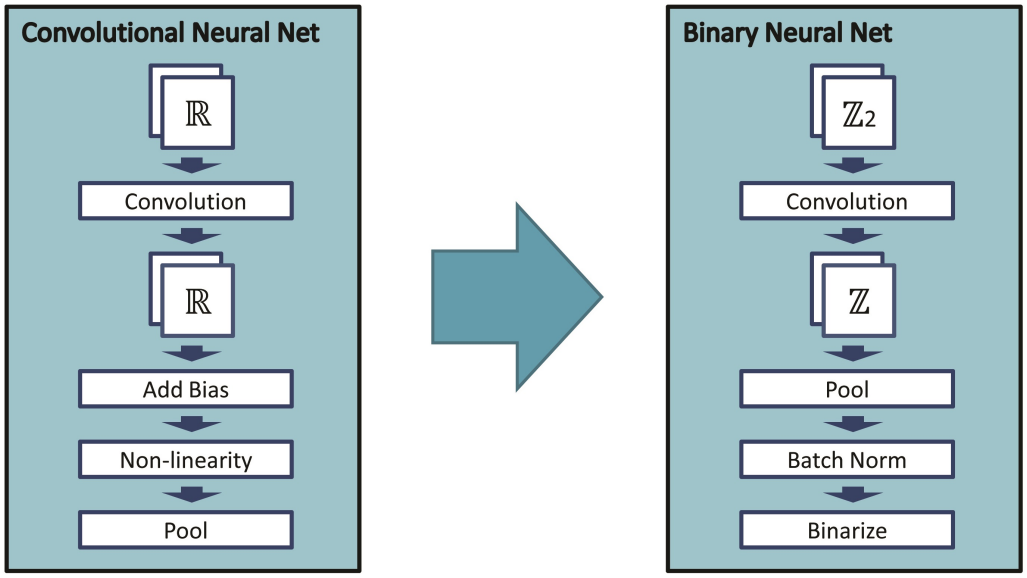

Abbildung 4.3: Grundlegende binäre Architektur im Vergleich mit einer regulären CNN Architektur, Quelle: [ZSZ+17]

Es wird zur FPGA-basierten Umsetzung die bereits in Kapitel 2.1 vorgestellte Struktur bestehend aus Faltungslagen gefolgt von Lagen zur Anwendung des Pooling-Verfahrens und zur Normalisierung der Daten eingesetzt. Hierbei wird gleichzeitig eine Binarisierung vorgenommen.
Die Systemarchitektur der Hardwareumsetzung ist in Grafik 4.4 dargestellt.

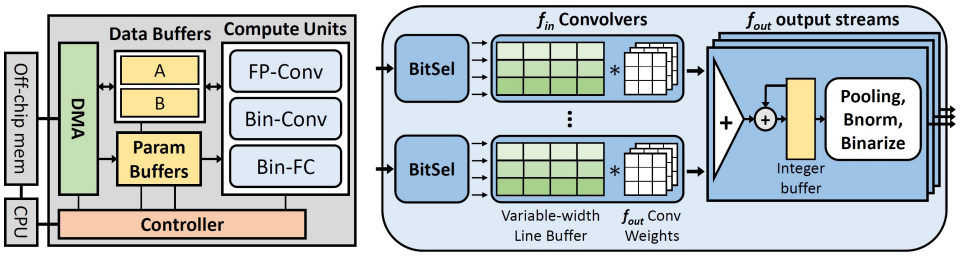

Abbildung 4.4: FPGA Umsetzung der binären Architektur, Quelle: [ZSZ+17]

Ähnlich wie bei der ZynqNet Implementierung werden hier Eingangsdaten und Gewichte von der ARM CPU zum Hardwaredesign übertragen. Die erste Faltungslage (FP-Conv) arbeitet mit Festkomma-Eingangsdaten und führt die Binarisierung durch. Dies wird durch die erste Faltung gefolgt von einer binären Batch-Normalisierung realisiert. Die nachfolgenden Lagen (Bin-Conv, Bin-FC) arbeiten nun mit binären Ein- und Ausgangsdaten. Die vorgestellten Hardwareelemente dieser Veröffentlichung werden als Basis für die spätere Umsetzung in dieser Arbeit benötigt und werden im Kapitel 7 noch im Detail erläutert. Tabelle 4.1 zeigt die exzellente Performanz des Systems im Vergleich zu CPU und GPU Implementierungen.

	Ausführungszeiten pro Bild (ms)			
Layer	**mGPU**	**CPU**	**GPU**	**FPGA**
Conv1	-	0,68	0,01	1,13
Conv2-5	-	13,2	0,68	2,68
FC1-3	-	0,92	0,04	2,13
Gesamt	90	14,8	0,73	5,94
Speedup	1,0x	6,1x	123x	15,1x
Power(Watt)	3,6	95	235	4,7
Bilder/Sek/Watt	3,09	0,71	5,83	35,8

Tabelle 4.1: Performanz und Energieeffizienz der binären Implementierung auf CPU, GPU und FPGA Plattformen im Vergleich, Quelle: [ZSZ+17]

Die binäre FPGA Implementierung benötigt nur 5,94ms pro Eingangsbild und ist damit 15,1 mal schneller als die Umsetzung auf der Nvidia Jetson TX1 GPU (mGPU). Im Vergleich zur Intel Xeon E5-2640 CPU ist die Hardwareumsetzung um den Faktor 2,5x schneller.

Lediglich die leistungsstarke Nvidia K40 Tesla GPU ist 8,1x schneller. Hinsichtlich der Performanz pro Watt liegt die FPGA-basierte Implementierung mit 35,8 Bildern pro Sekunde pro Watt deutlich vor allen anderen Umsetzungen.

4.3 Caffe to Zynq

Die Firma Xilinx befasst sich in der Abteilung reVision ebenfalls mit der effizienten Umsetzung von Neuronalen Netzen auf FPGAs. In der Präsentation aus [Kat17] wurden Binäre Neuronale Netze für den Einsatz auf Xilinx Zynq SoC vorgestellt. Dabei wird das Trainingsframework Caffe genutzt, um die vorgestellten Netze zu trainieren. Diese Arbeit stellt eine interessante Analyse des Einflusses einer Quantisierung auf das Trainingsergebnis auf, die in Tabelle 4.2 gezeigt ist.

Inference now 8 bit and below for maximum efficiency

Top-5 Accuracy	FP-32	FIXED-16 (INT16)	FIXED-8 (INT8)	Difference vs FP32
VGG-16	86.6%	86.6%	86.4%	(0.2%)
GoogLeNet	88.6%	88.5%	85.7%	(2.9%)
SqueezeNet	81.4%	81.4%	80.3%	(1.1%)

Tabelle 4.2: Analyse des Einflusses einer Quantisierung auf die Trainingsergebnisse, Quelle: [Kat17]

Es zeigt sich, dass durch die Verwendung einer 8-Bit Quantisierung die Trainingsergebnisse, besonders bei der VGG-16 Architektur, kaum beeinflusst werden. Im Vergleich zur 32-Bit Gleitkomma-Implementierung sinkt hier die Klassifikationsgenauigkeit lediglich um 0,2%.

Durch Anwendung einer Binarisierung soll die Performanz in Form von Bildern pro Sekunde pro Watt nach der in Tabelle 4.3 gezeigten Roadmap von den aktuell erreichten 19,0 (GoogLenNet) beziehungsweise 1,2 (AlexNet) auf über 50 Bilder pro Sekunde pro Watt gesteigert werden.

		May 2017	Roadmap
GoogLeNet @ batch = 1 3.2 Gops/img	Images/s	**114**	**370**
	Power (W)	6.0	7.0
	Images/s/watt	**19.0**	**52.9**

		May 2017	Roadmap
SSD @ batch = 1 62.4 Gops/img	Images/s	**6.3**	
	Power (W)	6.0	
	Images/s/watt	**1.1**	

		May 2017	Roadmap
FCN-AlexNet @ batch = 1 42.0 Gops/img	Images/s	**7.0**	
	Power (W)	6.0	
	Images/s/watt	**1.2**	

Tabelle 4.3: Xilinx reVision Roadmap für FPGA-basierte Neuronale Netze, Quelle: [Kat17]

Die Arbeit [Kat17] verdeutlicht die Vorteile von Binären Neuronalen Netzen und zeigt im Besonderen ihre Vorteile auf FPGA-basierten Systemen.

4.4 F-CNN: An FPGA-based Framework for Training Convolutional Neural Networks

Die Veröffentlichung [ZFL+16] stellt ein FPGA-basiertes Framework F-CNN zum Trainieren von Neuronalen Netzen vor. Im Gegensatz zu den vorherigen Arbeiten wird hier sowohl der Vorwärtspfad als auch der Rückwärtspfad der Berechnung umgesetzt (Forward/Backward Propagation). Dadurch kann neben der Klassifikation auch der vollständige Trainingsprozess in einem FPGA durchgeführt werden. Der Rückwärtspfad berechnet für ein gewünschtes Ausgangsergebnis die benötigten Gewichte. Dabei werden die Gewichte angepasst, bis sich ein optimales Ergebnis im Vergleich zur Referenz des Trainingsdatensatzes einstellt. Die Umsetzung fokussiert eine Handschrifterkennung auf Basis des MNIST Datensatzes durch eine LeNet-5 Netzarchitektur mit zwei Faltungslagen. Dabei wird ein heterogenes (Hybrid) System aus einer Stratix FPGA PCIe Karte und einem Desktop CPU System eingesetzt. Die Ergebnisse dieser Arbeit sind in der Tabelle 4.4 dargestellt.

Layers	N_k		F-CNN		Caffe (CPU)		Caffe (GPU)	
	Forward	Backward	Forward	Backward	Forward	Backward	Forward	Backward
L1(Conv)	12	12	0.59	1.21	6.84	7.70	2.55	7.19
L2(Pool)	24	24	0.53	0.57	1.85	0	0.04	0
L3(Conv)	12	12	4.67	10.32	36.37	33.11	5.33	3.38
L4 (Pool)	24	24	0.17	0.18	1.05	0	0.01	0
L5 (MLP)	24	24	0.92	1.82	2.04	2.02	0.05	0.4
L6 (MLP)	24	24	0.18	0.20	0.08	0.07	0.03	0.02
Total (Speedup)			21.4(4.3×)		91.1 (1×)		18.7 (4.9×)	
Power (Energy Efficiency)			27.3Watt (7.5×)				235Watt (1×)	

Tabelle 4.4: Ausführungszeiten des LENET-5 Trainings mit dem F-CNN Framework im Vergleich mit CPU und GPU, Quelle: [ZFL+16]

Es zeigt sich, dass das FPGA-basierte System für das Training (Vorwärts- und Rückwärtspfad) einen Speedup von 4,3x gegenüber der CPU Variante erreicht. Das GPU System, bestehend aus einer Nvidia Tesla K20x GPU, kann das Training sogar um den Faktor 4,9 schneller ausführen. In Bezug auf die Energieeffizienz ist das F-CNN Framework 7,5x sparsamer als die GPU Implementierung. Unter Berücksichtigung der sehr einfachen Netzarchitektur mit 2 Faltungslagen und der deutlich langsameren Ausführung der FPGA Variante bei tieferen Faltungslagen (L1 $\rightarrow$ L3) ist allerdings abzusehen, dass die Performanz dieses Frameworks für komplexere Klassifizierungsaufgaben deutlich abnimmt. Bei einer Bildklassifikation kommen üblicherweise 8 Lagen (AlexNet) oder 22 Lagen(GoogLeNet) bis hin zu 152 Lagen (ResNet) zum Einsatz [LJY17].

Kapitel 5

Heterogene Zielplattform

5.1 Auswahl einer heterogenen Zielplattform

Aus den vorhergehenden Kapiteln ist zusammenfassend zu entnehmen, dass FPGA-basierte Systeme für die Ausführung Neuronaler Netze Vorteile im Bezug auf die Performanz und Energieeffizienz bieten. Besonders im Vergleich zu einer CPU Variante zeigen sich deutlich verbesserte Ausführungszeiten. Bei der Umsetzung der Trainingsphase in Hardware hingegen konnten keine deutlichen Verbesserungen gegenüber einer GPU-basierten Implementierung festgestellt werden. Dies resultiert aus den stark optimierten DNN Softwarepaketen der Grafikkartenhersteller (z.B. Nvidia CUDA DNN) und der stetig wachsenden Performanz und Energieeffizienz von GPUs durch Verwendung kleinerer Chipstrukturen. Die Trainingsphase eines Neuronalen Netzes stellt im Gegensatz zur Ausführungsphase einen einmaligen Prozess zur Datenaufbereitung dar. Entsprechend kann man die Energieeffizienz in der Trainingsphase als zweitrangig betrachten. Eine Implementierung auf einem GPU-basierten heterogenen System bietet flexible Anpassungsmöglichkeiten durch reine Softwareimplementierung (CUDA, C++). Außerdem ist die Speicheranbindung zur GPU mit PCIe Schnittstelle und schnellem Onboard DDR5 Speicher besser für die Verarbeitung großer Trainingsdaten geeignet. Die Ausführung eines Neuronalen Netzes ist als Anwendung im Langzeitbetrieb ausgelegt und soll für den Einsatz auf mobilen Systemen nach Möglichkeit energieeffizient und performant operieren.

5.2 Design einer heterogenen Zielplattform

Entsprechend des oben ausgeführten Auswahlverfahrens der heterogenen Zielplattform wurde für die Umsetzung in dieser Arbeit eine Zielplattform bestehend aus CPU in Kombination mit GPU und einer eingebetteten Plattform bestehend aus CPU mit angeschlossener FPGA Partition entworfen. Bild 5.1 zeigt das Design der heterogenen Plattform.

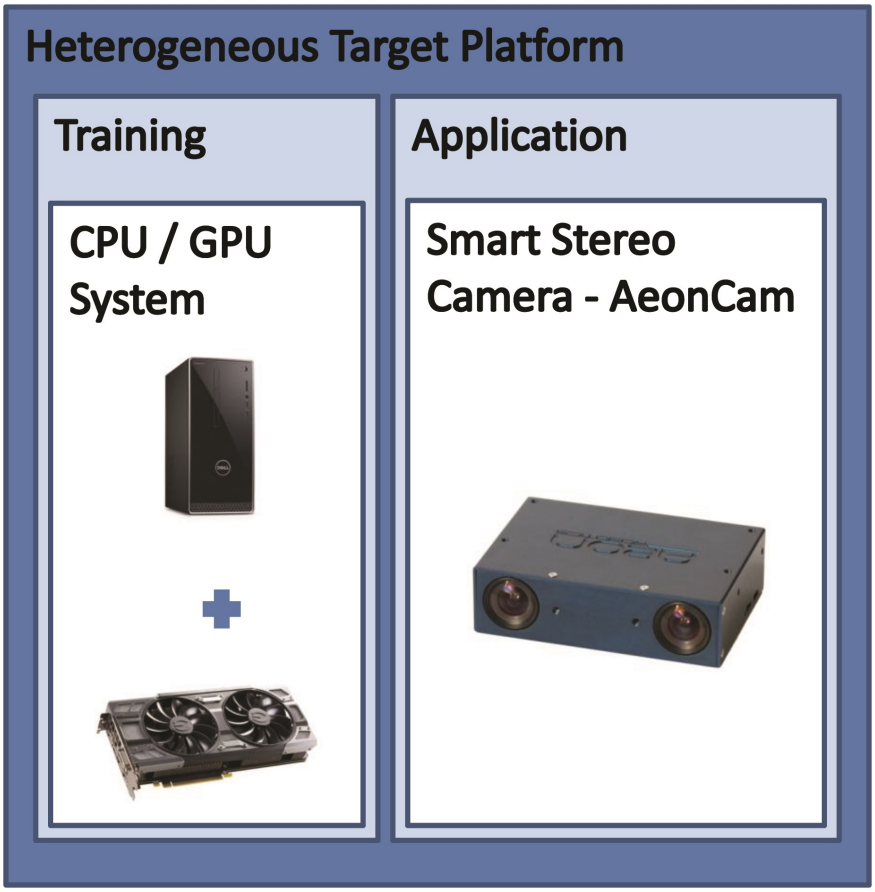

Abbildung 5.1: Systemdesign der heterogenen Zielplattform

In diesem Systemdesign ist ein heterogenes System bestehend aus einer CPU und GPU als Plattform zur Umsetzung der Trainingsphase vorgesehen. Für die Ausführung des Neuronalen Netzes wurde im Vorfeld speziell für diese Arbeit eine FPGA-basierte Stereokamera "AeonCam" entwickelt. Diese Plattform verfügt über einen integrierten ARM Prozessor, ein angebundenes FPGA und zwei Kameras zur Bildaufnahme. Dadurch stellt diese Plattform ein geschlossenes heterogenes System dar und kann durch die Stereokamera-Konfiguration Farb- und Tiefeninformation zur Umsetzung einer Objektklassifikation bereitstellen. Die AeonCam Plattform wird im Kapitel 7 zum Hardware- und Systemdesign im Detail beschrieben. Entsprechend der vorgestellten Optimierung aus Kapitel 3.1 soll zur effizienten Umsetzung ein Binäres Neuronales Netz implementiert werden.

Kapitel 6

Automatisiertes Lernen

6.1 Warum automatisiertes Lernen?

Ein Schwerpunkt dieser Arbeit ist es, den Trainingsprozess eines Neuronalen Netzes zur Objektklassifikation zu optimieren. Hierzu wurde der Gesamtprozess einschließlich Erstellung des Trainingsdatensatzes, Auswahl und Implementierung einer Netzarchitektur sowie Ausführung des Trainings und der Anpassung der Trainingsparameter manuell am Beispiel der Objektklassifikation durchgeführt und der Aufwand für jeden Teilschritt protokolliert. Die Ergebnisse dieser Analyse sind in Tabelle 6.1 dargestellt.

Vorgang	Zeitaufwand	Ressourcen
Erstellen des Datensatzes • Auswahl & Sortierung des Bildmaterials • Bildvorverarbeitung (Schneide, Skalieren,..)	~ 1-2 Wochen‖	~ 0,5 PM
Design der Netz-Architektur & Anpassung der Trainingsparameter	~ 1-2 Wochen‖	~ 0,5 PM
Training (CPU)*	267 Stunden	16,66kWh
Training (CPU + GPU)**	15 Stunden	2,59kWh

* Intel Core i7 6700 @ 3,4 GHz : Leistungsverbrauch 62,4W

** Intel CPU + GTX 970 GPU : Leistungsverbrauch 21,5W + 151W = 172,5W

‖ Basierend auf eigener Messung zur Erstellung von Datensatz und Netzarchitektur

Tabelle 6.1: Aufwand in Teilschrittanalyse eines Trainingsprozesses am Beispiel der Objektklassifikation

Die Untersuchung zeigt, dass die Schritte zur Erstellung eines Datensatzes und die Anpassungen der Netzarchitektur mit insgesamt 2-4 Wochen Arbeitszeit einen Großteil des Entwicklungsaufwandes entsprechen. Der Zeitaufwand für den Trainingsprozess ist hingegen, besonders mit GPU-Unterstützung,

mit einer Dauer von 15 Stunden eher zu vernachlässigen. Daher soll in dieser Arbeit ein Framework zur automatisierten Erstellung von Datensätzen mit abgestimmter Netzarchitektur implementiert werden. Zusätzlich sollen die Trainingsparameter wie Batch-Größe, Lernrate und Trainingsepochen speziell auf die Anwendung des Automatisierten Lernens einer Objektklassifikation abgestimmt werden. Im folgenden Abschnitt wird die technische Umsetzung dieser Ziele erläutert.

6.2 Framework zur Automatisierung des Lernprozesses

Die Idee des Automatisierten Lernens kann im speziellen Fall der Objektklassifikation durch eine Bildersuche im Internet realisiert werden. Dadurch, dass verschiedene Anbieter von Internet-Suchmaschinen zu einem Suchbegriff einer Objektklasse über eine Bildersuche Bildmaterial bereitstellen, können diese Informationen zu einem Trainingsdatensatz zusammengestellt werden. Daraus ist das in Bild 6.1 skizzierte Framework zum Automatisierten Lernen von Binären Neuronalen Netzen (AL BNN) dieser Arbeit entstanden.

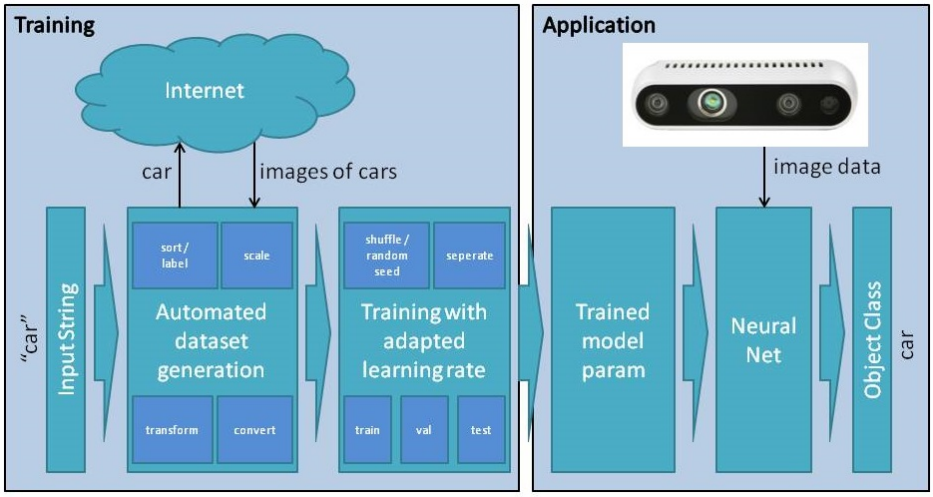

Abbildung 6.1: Framework zur Automatisierung des Lernprozesses

Das AL BNN Framework ist in der Programmiersprache Python implementiert, um die Module zur automatisierten Generierung des Datensatzes und das Trainingsframework direkt miteinander verbinden zu können. Zur Generierung des Datensatzes erwartet das zuständige Modul als Eingangsinformation eine Liste zu trainierender Objekte in der Form

```
labels = ['bottle', 'hammer', 'mug', 'scissor', 'headphone', 'teddy']
```

Diese Informationen werden zusammen mit den Angaben zur Konfiguration des Trainingsprozesses im Top-Level Modul train_al_bnn.py eingetragen. Die Sektion enthält folgende zu konfigurierende Parameter:

```
#------ CONFIGURATION SECTION START ------------------

#dataset/network parameters
labels = ['bottle', 'hammer', 'mug', 'scissor', 'headphone', 'teddy']
num_classes = 10
images_per_class = 300
resolution = 64 # set to 32 => 32x32, 64 => 64x64, 128 => 128x128

#training parameters
conv_layer = 6  # conv layer, select 6, 8 or 10
batch_size = 50
num_epochs = 500
LR_start = 0.0001
LR_fin   = 0.00003

#dataset & output path
dataset_path = '/path/to/Automated-Learning-BNN/dataset'
output_path = '/path/to/Automated-Learning-BNN/output'

#------ CONFIGURATION SECTION END ------------------
```

Dabei entspricht der Parameter “num_classes” der maximalen Anzahl möglicher Klassen. Dieser Wert muss nicht zwingend der Anzahl der Elemente in “labels” entsprechen. Die Restriktion der maximalen Anzahl von Klassen ist für die Hardwareumsetzung im Verlauf dieser Arbeit notwendig. In dieser Konfiguration können bis zu 10 verschiedene Objektklassen trainiert werden. Der Parameter “images_per_class” gibt an, wie viele Bilder der jeweiligen Klasse generiert werden sollen. Dabei gibt die Angabe “resolution” die gewünschte Auflösung der Bilder vor. Entsprechend soll in dieser Konfiguration ein Datensatz bestehend aus sechs Klassen mit jeweils 300 Bildern mit der Auflösung von 64x64 Bildpunkten generiert werden. Das iterative Verfahren des AL BNN Frameworks zum Zusammentragen und Aufbereiten des Bildmaterials ist in Darstellung 6.2 als Programmablaufplan verdeutlicht.
Über die Funktion “generateDatatset” werden diese Information zusammen mit dem Pfad zur Speicherung an das Software-Modul zur automatisierten Generierung des Datensatzes weitergereicht. Im folgenden Schritt wird über die Funktion “createObjectClass” zunächst am angegebenen Speicherort des Datensatzes ein Unterordner für jede Klasse erstellt und über die Funktion “getImages” mit Bildmaterial gefüllt. Die Funktion “getImages” benutzt hierfür den Selenium Webdriver für den Mozilla Firefox Webbrowser oder den Google Chrome Webbrowser, um über die Google API zur Bildersuche mehrere Anfragen zum Herunterladen von Bildmaterial zu sen-

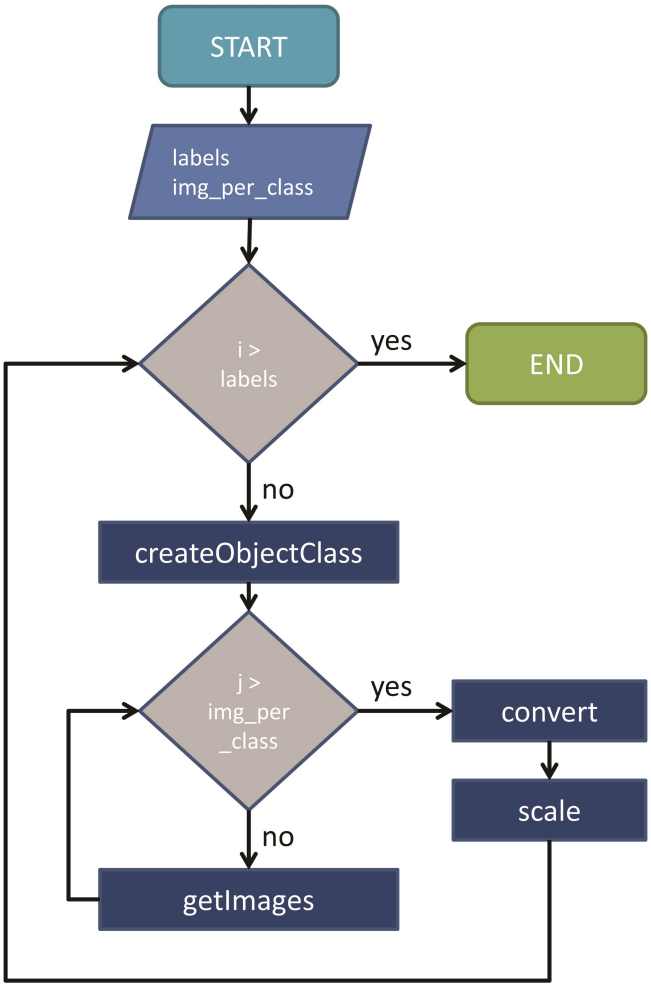

Abbildung 6.2: Programmablaufplan der Datensatz-Generierung

den. Dabei werden die Anfragen für Bilder mit den Endungen [.jpg, .png, .jpeg] für die entsprechende Klasse wiederholt, bis die Anforderungen des "images_per_class" Parameters erfüllt sind. Anschließend werden die Bilder jeder Klasse über das Bildverarbeitungstool "mogrify" als Teil des ImageMagick Programms zu einem einheitlichen Datensatz formatiert. Hierfür werden über den Befehl

```
p = subprocess.Popen('find ' + path + '/"' + name + '"' + ' -name [
    filename] -exec [mogrify -cmd {} \;', shell=True).wait().
```

blockende Subprozesse für jeden Verarbeitungsschritt gestartet. In dieser Form werden folgende Verarbeitungsschritte ausgeführt:

- Umwandlung von Bildern mit .png Format in das JPG Dateiformat
- Konvertierung in den einheitlichen Farbraum sRGB
- Veränderung jedes Bildtyps zur TrueColor Repräsentation
- Skalierung der Auflösung zur gewünschten einheitlichen Auflösung [z.B. 64x64]

Nach Ausführung der "generateDatatset" Funktion ergibt sich somit die Datensatz-Struktur aus Bild 6.3 für den neu generierten Datensatz aus der Beispielkonfiguration.

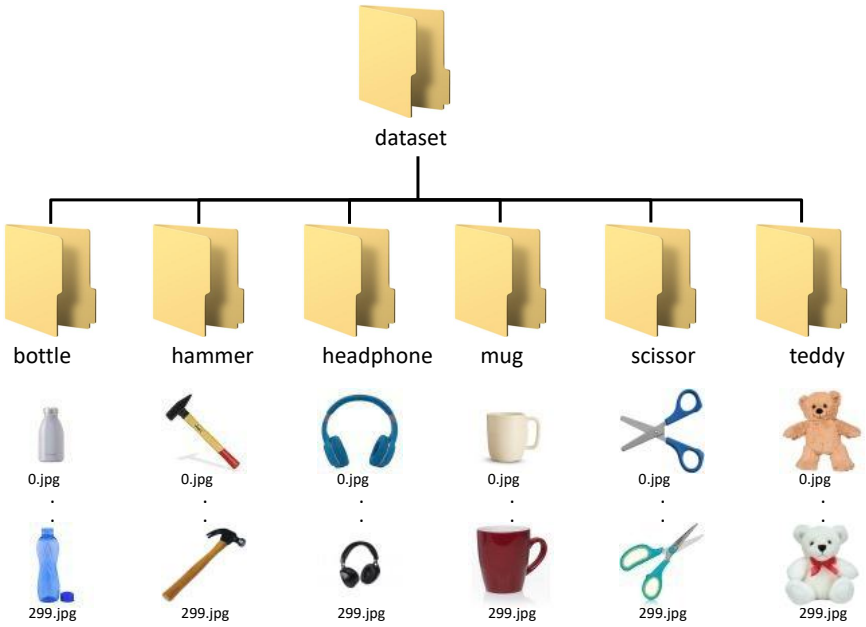

Abbildung 6.3: Ordnerstruktur des automatisch generierten Datensatzes

Die Unterordner für jede Klasse enthalten n= images_per_class Bilder im JPEG sRGB/TrueColor Format. In der Beispielkonfiguration entspricht dies einem Datensatz bestehend aus 6 x 300 = 1800 Bildern. Im Vergleich zum CIFAR-10 Datensatz mit 6000 Bildern pro Klasse wird die stark reduzierte Datenmenge deutlich. Um trotzdem eine Klassifikation mit ausreichend

genauer Präzision zu realisieren, wird das Trainingsframework speziell auf die Gegebenheiten der automatisch generierten Datensätze angepasst. Diese Anpassungen werden im nächsten Abschnitt erläutert.

6.2.1 Framework zum Trainieren Binärer Neuronaler Netze

Als Trainingsframework wurde nach [CHS+16] das Python-basierte Framework "Theano" zur Umsetzung des Trainingsprozesses ausgewählt. Diese Umgebung kann als Python-Programm direkt in die Struktur des AL BNN Frameworks integriert werden und unterstützt GPU Beschleunigung durch CUDA/OpenCL Support. In Kombination mit dem "Lasagne"-Framework können sowohl die Trainingsparameter als auch eigene spezialisierte Netz-Lagen implementiert und weitreichend angepasst werden. Dabei wird die in Kapitel 3 vorgestellte Struktur zum Trainieren von Binären Neuronalen Netzen auf Basis des CIFAR-10 Datensatzes als Grundstruktur verwendet.

6.2.2 Grundstruktur des Binären Neuronalen Netzes

Die Struktur des Binären Neuronalen Netzes basiert auf dem Aufbau in Grafik 6.4.

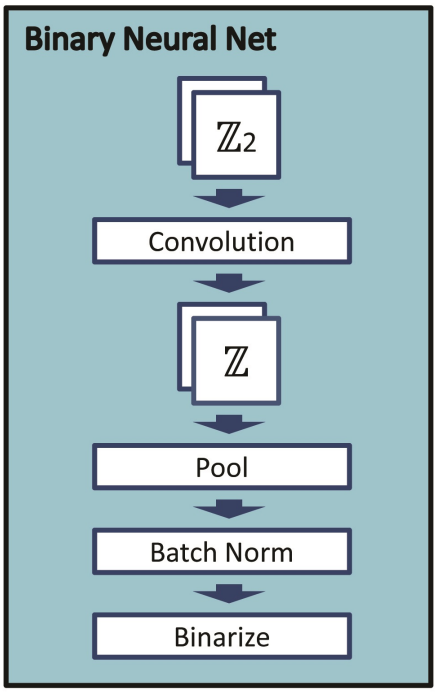

Abbildung 6.4: Grundstruktur des Binären Neuronalen Netzes

Zunächst wird eine Faltungslage (Convolution), wie in Kapitel 2.1 vorgestellt, angewendet. Dabei wird die im "Lasagne" Framework bereits verfügbare Klasse der Faltungslage lasagne.layers.Conv2DLayer überladen und um einen Abschnitt für die binäre Faltungsberechnung erweitert:

```
def convolve(self, input, deterministic=False, **kwargs):
        self.Wb = binarization(self.W,self.H,self.binary,deterministic
            ,self.stochastic,self._srng)
        Wr = self.W
        self.W = self.Wb
        rvalue = super(Conv2DLayer, self).convolve(input, **kwargs)
        self.W = Wr
        return rvalue
```

Zur Binarisierung der Gewichte wird hier der deterministische Ansatz aus Kapitel 3.1.2 verwendet:

```
    def binarization(W,H,srng=None):
        # [-1,1] -> [0,1]
        Wb = hard_sigmoid(W/H)
        Wb = T.round(Wb)
        # 0 or 1 -> -1 or 1
        Wb = T.cast(T.switch(Wb,H,-H), theano.config.floatX)
    return Wb
```

Dabei werden die Koeffizienten zur Initialisierung der Gewichte aus [GB10] verwendet. Als Aktivierungsmethode wurde hierbei die Linear / Identity-Funktion entsprechend

$$\phi(x) = x \tag{6.1}$$

angewendet.
Das gleiche Verfahren wird bei den Dense-Lagen (Dense Layer / Fully-Connected) angewendet. Die gezeigte Pooling-Lage entspricht der Implementierung der lasagne.layers.MaxPool2DLayer Klasse. Zur binären Normalisierung wird die Klasse aus lasagne.layers.BatchNormLayer angewendet und nachfolgend eine tanh Aktivierung des Typs

$$\phi(x) = \tanh(x) \tag{6.2}$$

vorgenommen.

6.2.3 Einbindung in das AL BNN Framework

Durch den ersten Schritt des automatisierten Lernens konnte ein Datensatz in Form von Bildmaterial in einer klassenbasierten Ordnerstruktur automatisiert erstellt werden. Für den anschließenden Trainingsprozess müssen diese Daten entsprechend der Ordnerstruktur eingelesen und mit Labels versehen

werden. Außerdem müssen die Daten gemischt vorliegen und in Trainings-, Validierungs- und Testdatensätze aufgeteilt werden. Dieser Schritt ist im Bild 6.5 als Modulblock des Trainingsprozesses (train_al_bnn) dargestellt.

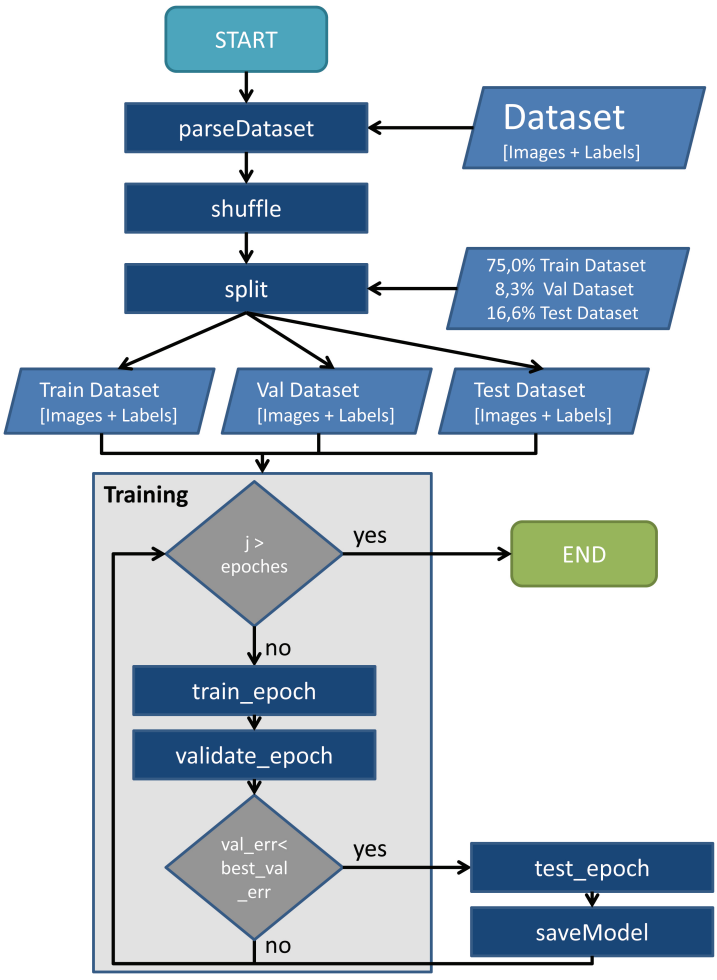

Abbildung 6.5: Programmablaufplan des Trainingsprozesses (train_al_bnn)

Um den Schritt der Datenvorbereitung effizient auszuführen, werden in der Funktion “parseDataset” zunächst die Pfade zu jedem Bild aufgelistet. Die “shuffle”-Funktion aus dem eingebundenen sklearn Framework übernimmt dabei das Mischen der Pfade. Diese Liste wird nun nach den Vorgaben (16,6% Test, 8,3% Validierung) in die entsprechenden Datensätze aufgeteilt. Im Anschluss wird das Bildmaterial über die Funktion loadImageAndTarget nach den Pfadangaben in den Datensatz eingelesen und mit einem Label entsprechend des Ordnernamens versehen. Die Bilddaten liegen nun in den Strukturen [train, valid, test]_X vor. Die zugehörigen Labelreferenzen sind

in [train, valid, test]_y abgelegt.
Die Bilddaten werden über

```
train_X = np.reshape(np.subtract(np.multiply(2./255.,train_X),1.)
    ,(-1,3,resolution,resolution))
valid_X = np.reshape(np.subtract(np.multiply(2./255.,valid_X),1.)
    ,(-1,3,resolution,resolution))
test_X = np.reshape(np.subtract(np.multiply(2./255.,test_X),1.)
    ,(-1,3,resolution,resolution))
```

auf den Wertebereich [-1,+1] skaliert und in die von Theano vorgesehene Datenstruktur (-1,input channels,resolution,resolution) formatiert. Die Referenz-Label werden über die Funktionen "hstack" und "eye" aufbereitet, um die Referenzen in eine 1-dimensionale Struktur mit optimierter 1-hot Enkodierung umzuformen. Außerdem wird über eine Multiplikation mit 2 und Subtraktion von 1 die "hinge loss" Funktion nach

$$l(x) = \max(0, 1 - t \cdot y) \tag{6.3}$$

des Ausgangs realisiert.

6.2.4 Trainingsprozess des AL BNN Frameworks

Zur Anpassung der Gewichte während des Trainings wird die Berechnung der adaptiven Lernratenanpassung über die "Adaptive Moment Estimation"-Funktion (Adam) aus [KB14] durchgeführt. Es werden zunächst die Gradienten über den Aufruf

```
W_grads = binary_net.compute_grads(loss, bnn)
```

bestimmt.
Dies entspricht der Bestimmung des Gradientenabstiegs für die gemittelten letzten Gradienten m_t und der quadrierten letzten Gradienten v_t nach

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \qquad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{6.4}$$

Daraus können die Momente nach

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{6.5}$$

bestimmt werden, um anschließend über Anwendung der Adam-Funktion

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{6.6}$$

die Modellparameter anzupassen. Über die Funktion “clipping_scaling” wird abschließend die Skalierung der Gewichte im Bezug auf die Lernrate nach [GB10] mit dem Skalierungsfaktor W_LR_scale vorgenommen.

```
W_LR_scale = 1 /sqrt(1.5 / (n_inputs + n_filter)))
```

6.2.5 Anpassung der Lernrate und der Datensatzgröße

Um ein optimales Trainingsergebnis unter Verwendung des automatisch generierten Trainingsdatensatzes zu erzielen, wurden ausführliche Testreihen mit unterschiedlichen Datensatzgrößen und Lernraten durchgeführt. Für die Datensatzgenerierung wurden Größen von 50 bis 1200 Bildern pro Klasse evaluiert. Die Lernrate wurde dabei adaptiv über Verwendung der Adam-Funktion in Verbindung mit einer abnehmenden Lernrate nach

$$LR* = (LR_{fin}/LR_{start}) * *(\frac{1}{n_{epochs}}) \tag{6.7}$$

angepasst. Dabei entsprechen LR_{fin}, LR_{start} und n_{epochs} Hyperparametern, die während der Testreihen optimiert wurden. So wurden folgende Lernrate Konfigurationen getestet:

$$LR_{start} = 0.1 \rightarrow LR_{fin} = 0.0001 \tag{6.8}$$
$$LR_{start} = 0.01 \rightarrow LR_{fin} = 0.0001 \tag{6.9}$$
$$LR_{start} = 0.001 \rightarrow LR_{fin} = 0.0001 \tag{6.10}$$
$$LR_{start} = 0.003 \rightarrow LR_{fin} = 0.00003 \tag{6.11}$$
$$LR_{start} = 0.0001 \rightarrow LR_{fin} = 0.000003 \tag{6.12}$$
$$LR_{start} = 0.0001 \rightarrow LR_{fin} = 0.000003 \tag{6.13}$$
$$LR_{start} = 0.00001 \rightarrow LR_{fin} = 0.000003 \tag{6.14}$$
$$LR_{start} = 0.00001 \rightarrow LR_{fin} = 0.0000003 \tag{6.15}$$

Als Ergebnis der Testreihen wurde eine Datensatzgröße von 300 Bildern pro Klasse als guter Kompromiss zwischen zu wenig Trainingsdaten und nicht klassenspezifischen Suchergebnissen gefunden. Für diese Datensatzgröße wurden die Parameter zur Anpassung der Lernrate mit den Werten

```
batch_size = 50
num_epochs = 500
LR_start = 0.0001
LR_fin   = 0.00003
```

als optimale Trainingsparameter bestimmt.

6.2.6 Visualisierung des Trainingsprozesses

Der Verlauf des Trainingsprozesses ist zur Optimierung der Trainingsparameter besonders wichtig. Da nach jeder Epoche des Trainings (Durchlauf des Datensatzes) zur Validierung eine Probe der Klassifizierung anhand des Validierungsdatensatzes nach 3.1.3 vorgenommen wird, können diese Resultate als Zwischenergebnis des Trainings ausgewertet werden. Das AL BNN Framework wurde so konzipiert, dass es neben den Log-Ausgaben in der Konsolendarstellung und einer Log-Textdatei auch eine grafische Darstellung des Trainingsverlaufs erstellt. Hierzu werden die Zwischenergebnisse der Klassifizierungsgenauigkeit und des Trainingsverlustes als Funktion über die Trainingsepochen aufgezeichnet. Das erstellte Diagramm ist in Abbildung 6.6 gezeigt.

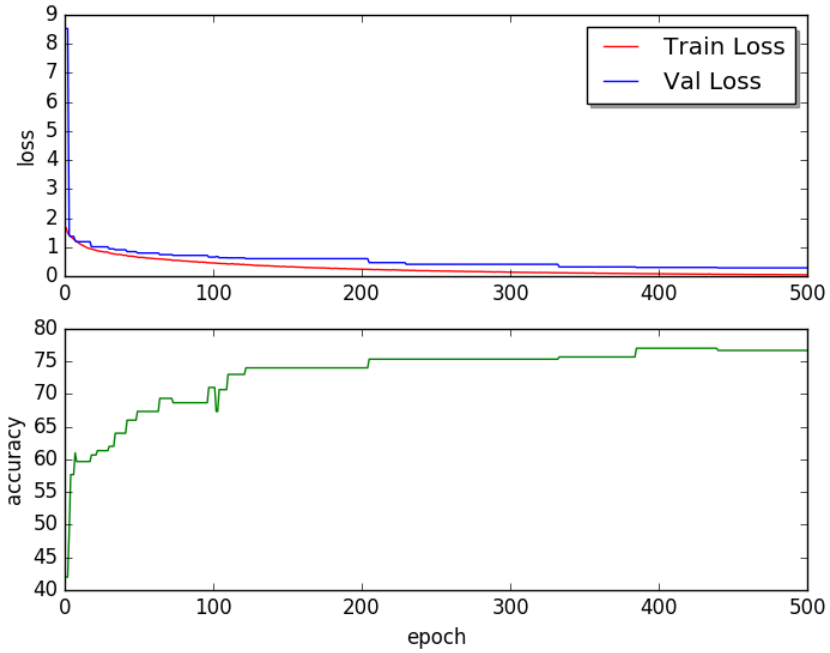

Abbildung 6.6: Grafische Darstellung des Trainingsprozesses über Klassifikationsgenauigkeit und Trainingsverlust

In dieser Abbildung ist der Trainingsverlauf in grafischer Darstellung am Beispiel eines Binären Neuronalen Netzes mit einer $32x32$ Eingangsauflösung und 9 Netzlagen (6 Convolutional Layer, 3 Dense Layer) gezeigt. So kann während des Trainingsprozesses der Verlauf überwacht und über Veränderung der Trainingsparameter (Epochen, Lernrate, Batch-Größe) gegebenenfalls angepasst werden.

Das AL BNN Framework nutzt zusätzlich die Zwischenergebnisse unter Verwendung des Testdatensatzes, um das Ergebnis der Klassifikation in eine sogenannte Konfusionsmatrix (siehe Bild 6.7) einzutragen. Hier sind die Objektklassen als Aussage der trainierten Klassifikation und den Referenz-Labeln aus dem Testdatensatz gegenübergestellt. Hierdurch kann direkt das Klassifikationsergebnis für die jeweilige Objektklasse getrennt voneinander betrachtet werden. Auch diese Darstellung wird während des Trainingsprozesses stetig aktualisiert, so dass eine direkte Aussage über den aktuellen Trainingsfortschritt getroffen werden kann.

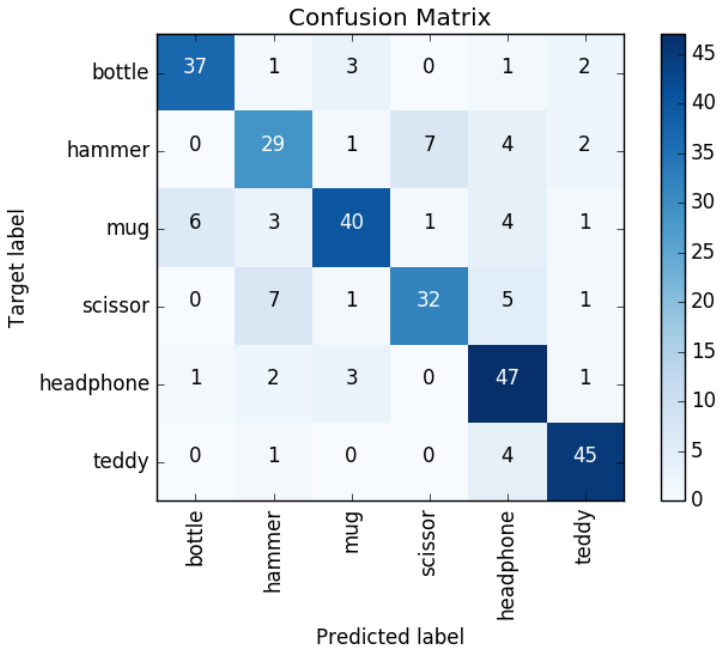

Abbildung 6.7: Darstellung des Trainingsprozesses als Konfusionsmatrix

In dieser Darstellung wird deutlich, dass die Klassifikationsgenauigkeit der Objektklassen “headphones” und “teddy” mit über 90% sogar deutlich höher ausfällt als das Ergebnis der Gesamtklassifikation aus Grafik 6.6. Die Klassifikationen der Objekte “hammer” und “scissor” sind hingegen oft vertauscht und sind somit schwierig voneinander zu unterscheiden.

6.3 Auswahl einer geeigneten Netzarchitektur

Zusätzlich zur Optimierung des Trainingsprozesses wurde eine optimale Netzarchitektur zur Objektklassifikation auf Basis der automatisch generierten

Datensätze gesucht. Hierzu wurden die in Bild 6.8 dargestellten Architekturen für eine genauere Untersuchung ausgewählt und implementiert. Des Weiteren wurde der Einfluss verschiedener Auflösungen der Eingangsbilder mit $32x32$, $64x64$ und $128x128$ Bildpunkten evaluiert.

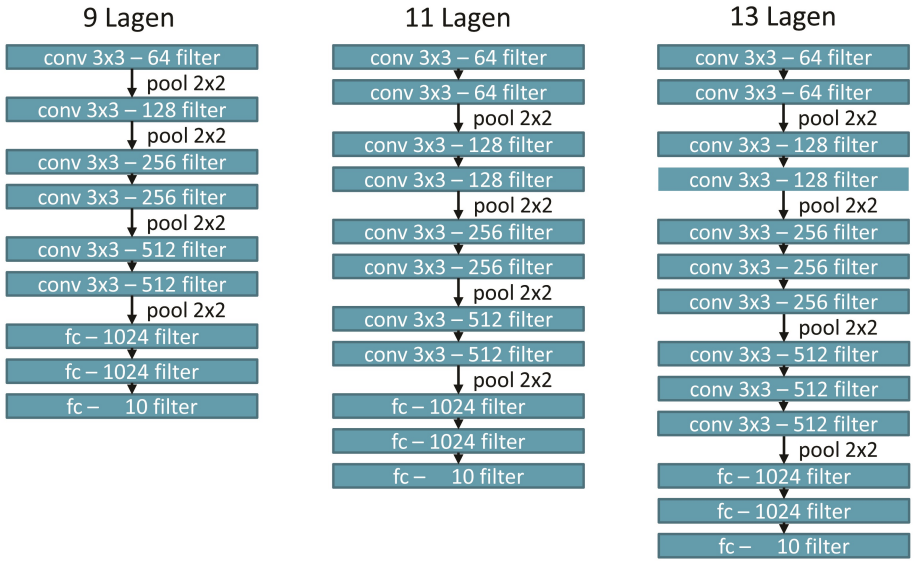

Abbildung 6.8: Implementierte Netzwerkarchitekturen für die Evaluation

Die evaluierten Eingangsauflösungen sind in Grafik 6.9 gezeigt.

Abbildung 6.9: Getestete Eingangsauflösungen

Die Untersuchungen wurden bereits zu diesem Zeitpunkt durch das AL BNN Framework deutlich vereinfacht, da hier die Datensätze und die Netzwerkarchitekturen konfiguriert und der Trainingsprozess automatisiert ablaufen konnte. Die Ergebnisse dieser Evaluation in Form von Klassifikationsgenauigkeit nach dem Trainingsprozess sind in Tabelle 6.2 ausgeführt.

	Architektur			
Auflösung		9 Lagen	11 Lagen	13 Lagen
	32 x 32	76,7%	72,7%	73,7%
	64 x 64	82,3%	75,3%	72,0%
	128 x128	77,3%	77,0%	75,7%

Tabelle 6.2: Ergebnis der Evaluation verschiedener Eingangsauflösungen und Netzarchitekturen

Die Netzkonfiguration mit einer Eingangsbildauflösung von $64x64$ Pixeln und einem Lagenaufbau mit 9 Lagen hat somit in der Evaluation die besten Klassifikationsergebnisse gezeigt und wird deshalb im weiteren Verlauf dieser Arbeit als Netzkonfiguration gewählt.

6.4 Evaluation zur Verwendung von Tiefeninformationen

Ein weiteres Ziel dieser Arbeit ist die generelle Verbesserung der Klassifikation bei reduzierten Trainingsdaten. Deshalb wird im folgenden Abschnitt eine mögliche Optimierung der Eingangsdaten unter Verwendung von Tiefeninformationen untersucht und diskutiert. Dies ist besonders für die spätere Umsetzung auf der AeonCam Plattform interessant, da hier bereits Tiefeninformation bereitgestellt werden können. Der erste Ansatz zur Verwendung von Tiefeninformation in Neuronalen Netzen ist in Bild 6.14 gezeigt.

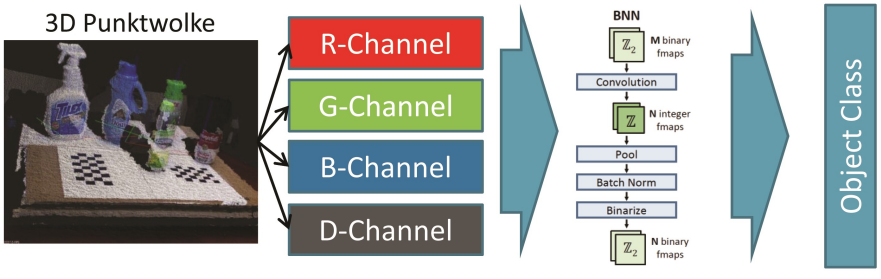

Abbildung 6.10: Verwendung von Tiefeninformation in Neuronalen Netzen

Hierbei werden die ursprünglichen RGB-Eingangsdaten ([Rot, Grün, Blau] Kanäle) um einen vierten Kanal erweitert. Die Verwendung des Tiefenkanals (D-Channel) beeinflusst die Eingangsdimensionen der ersten Lage des Neuronalen Netzes. So erweitert sich das Eingangsvolumen der Bilder von $64x64x3$ auf $64x64x4$. Neben den Änderungen der Implementierung müssen allerdings auch neue Trainingsdaten mit eingebundenen Tiefeninformationen erstellt werden. Hierfür wurde der Ansatz aus [He17] angewendet, um den bekannten CIFAR-10 Datensatz über Tiefenschätzung eines weiteren Neuronalen Netzes zu einem neuen Datensatz zu transformieren. Die ursprünglichen Dateien des CIFAR-10 Datensatzes (6 Dateien mit 1024x1024 Bildern) wurden entsprechend entpackt und als Einzelbilder abgelegt. Anschließend wurde das in [LSL15] vorgestellte Neuronale Netz zur Schätzung der Tiefeninformation aus monokularen Bildern angewendet. Hierfür wurde das vortrainierte Modell in einer MATLAB Implementierung ausgeführt. Das Ergebnis dieser Umformung ist in Abbildung 6.11 dargestellt.

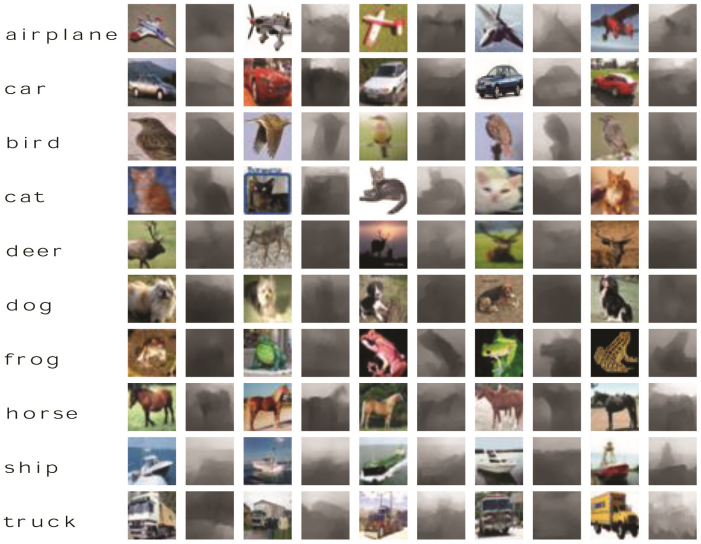

Abbildung 6.11: Tiefenschätzung für Bilder des CIFAR-10 Datensatzes

Als weiterer Ansatz wurde ein eigener Datensatz aus einem RGBD-Datensatz aus [LBRF11] erstellt. In dieser Veröffentlichung wurden verschiedene Objekte über eine Microsoft Kinect Tiefenkamera aufgenommen und dabei über eine Rotationsplattform bewegt. Dadurch konnten Tiefeninformationen und Farbbilder der Objekte aus verschiedenen Blickwinkeln über 360° Rotationen aufgezeichnet werden. Es wurden Videosequenzen mit einer Bildwiederholrate von 20Hz bestehend aus 250 Bildern erstellt. Für jede der 51 Objektklassen wurden mehrere Objekte als Sequenz aufgenommen. So ergibt sich eine Gesamtgröße des Datensatzes von 250.000 RGB-D Bildern

der Auflösung 640x480. Zur Umsetzung der Evaluation der Tiefeninformationen für diese Arbeit wurden 10 spezifische Objekte [apple, ball, banana, bell pepper, mug, lemon, orange, pear, soda can, tomato] des Datensatzes ausgewählt und für die weitere Verwendung an die Struktur des CIFAR-10 Datensatzes angeglichen. So wurden einige Frames der Videosequenzen übersprungen, um lediglich Blickwinkeländerungen im Abstand von 5° abzubilden. Zusätzlich wurde das Bildmaterial auf einen Ausschnitt der Auflösung 64x64 Bildpunkte reduziert und die Tiefeninformationen normiert. Das Ergebnis dieser Umformung ist ein Datensatz bestehend aus 10 Objektklassen mit jeweils 2400 Bildern pro Klasse. So ergibt sich ein Datensatz bestehend aus 24.000 Einzelbildern mit zugehörigen Tiefenbildern. Ein Ausschnitt aus diesem RGB-D Datensatz ist in 6.12 gezeigt.

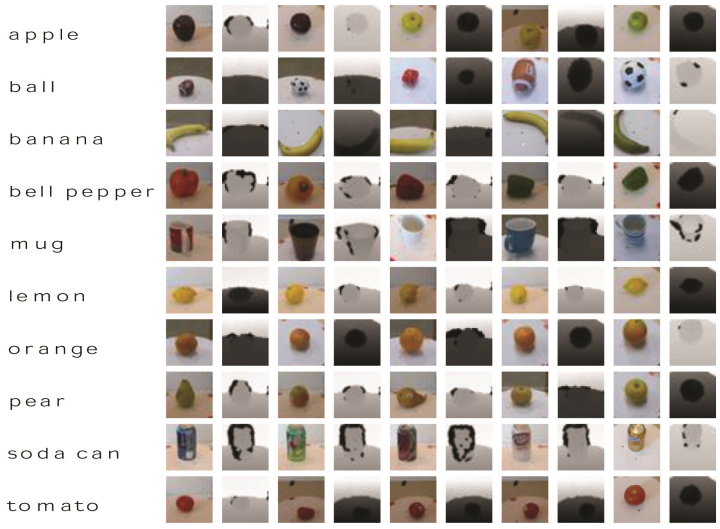

Abbildung 6.12: RGB-D Bilder des eigenen Datensatzes

Auf Basis dieser zwei Datensätze wurde die Evaluation mit der Trainingsumgebung des AL BNN Frameworks durchgeführt. Tabelle 6.3 stellt die Klassifikationsergebnisse der beiden Versuche als Trainingsergebnisse mit und ohne Verwendung von Tiefeninformationen gegenüber.
Aufgrund der Ergebnisse der vorgestellten Evaluation konnte keine Verbesserung der Klassifikation durch Verwendung von Tiefeninformationen festgestellt werden. Sowohl die geschätzten Tiefeninformationen, als auch das speziell erstellte “Custom Dataset” zeigen eine leicht erhöhte Fehlerrate der Klassifikationen. Dies kann im Fall des Custom Datasets durch die geringe Tiefenauflösung des verwendeten Sensors, besonders bei der gewählten Entfernung der aufgenommenen Objekte, erklärt werden. Unter Berücksichtigung des hohen Aufwands zur Erstellung eines speziellen RGB-D Datensat-

Eingangsdaten	Test Ergebnis (Err %)
CIFAR10 (RGB)	11,77%
CIFAR10 (RGB + Estimated Depth)	11,82%
Custom Dataset (RGB)	13,03%
Custom Dataset (RGBD)	15,53%

Tabelle 6.3: Trainingsergebnisse mit und ohne Tiefeninformationen

zes und der reduzierten Klassifikationsgenauigkeit in der Evaluation werden für die weitere Umsetzung in dieser Arbeit keine Tiefeninformationen zur Verbesserung des Trainingsergebnisses eingesetzt.

6.4.1 Tiefensegmentierung als Vorverarbeitung der Eingangsdaten

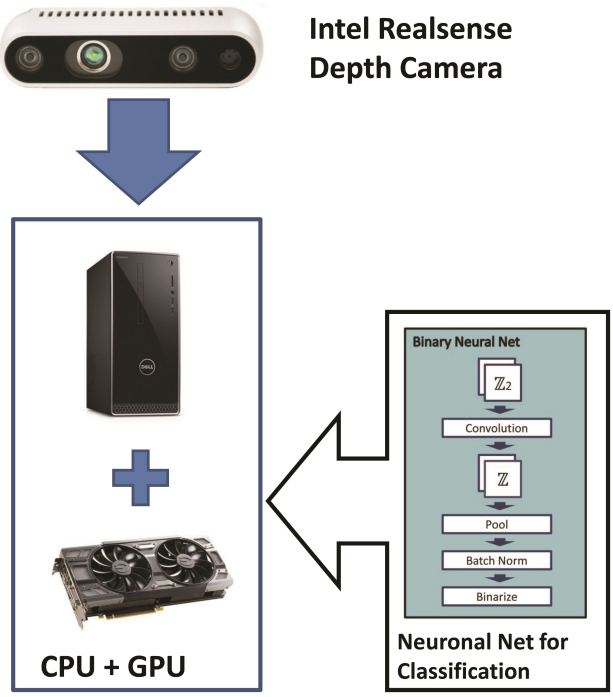

Abbildung 6.13: Aufbau mit Tiefenkamera-Sensor zur Segmentierung der Eingangsbilder

Im Gegensatz zur vorhergehenden Evaluation wurde auch die Verwendung von Tiefeninformationen zur Ausführungszeit analysiert. Dazu wurde zur Aus-führung das mit dem AL BNN trainierte Modell geladen. Als Eingangsdaten wurden die Bildinformationen eines Intel Realsense Stereo-Kamerasensors verwendet. Der Aufbau für diese Testreihe ist in Grafik 6.13 gezeigt. Über die Python-Variante der librealsense Bibliothek (pyrealsense) können die Farbbilder und die Tiefeninformationen des Sensors ausgelesen werden. Um die Tiefeninformationen entsprechend der Farbinformationen auszurichten, werden die Kalibrierungsparameter über

```
depth_sensor = profile.get_device().first_depth_sensor()
depth_scale = depth_sensor.get_depth_scale()
```

gelesen und über

```
align_to = rs.stream.color
align = rs.align(align_to)
```

entsprechend transformiert. Anschließend können die Tiefeninformationen genutzt werden, um eine einfache Tiefensegmentierung über eine Schwellwertsegmentierung nach den Vorgaben (clipping_distance) vorzunehmen. So wird über den Aufruf

```
bg_removed = np.where((depth_image_3d > clipping_distance) | (
    depth_image_3d <= 0), white_color, color_image)
```

der Hintergrund des Eingangsbildes mit weißen Farbwerten überschrieben. Ebenso werden Bildpunkte ohne zugehörige Tiefeninformationen (depth $\leq$ 0) überschrieben. Dadurch wird eine Tiefensegmentierung wie in Bild 6.14 dargestellt umgesetzt.

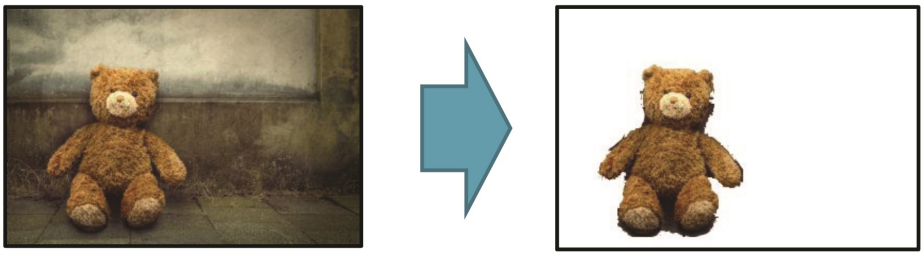

Abbildung 6.14: Beispiel einer Tiefensegmentierung

Anschließend können die segmentierten Bilder zu der gewünschten Eingangs-

größe des Neuronalen Netzes durch Definition einer Region (Region of Interest, "roi") und Skalierung auf 64x64 Pixel umgeformt werden. Da die Kanäle des Eingangsbildes in einer anderen Darstellung als die Trainingsdaten vorliegen, muss zusätzlich die Anordnung der Kanäle durch

```
x = x.transpose(2,0,1)
```

angepasst werden.
Zuletzt müssen die Bildinformationen noch, wie schon zur Vorbereitung der Trainingsdaten, über

```
x = np.reshape(np.subtract(np.multiply(2./255.,x),1.),(-1,3,res,res))
```

binarisiert und in die Datenstruktur [-1,num_channels,resolution,resolution] umgewandelt werden.
Für diese Eingangsdaten kann das AL BNN Framework nun eine Objektklassifikation vornehmen. Um die Vorteile der Tiefensegmentierung für die Objektklassifikation zu analysieren, wurde eine Testreihe mit 6 Testobjekten unterschiedlicher Klassen durchgeführt. Hierfür wurden jeweils die Ergebnisse der Klassifikation mit weißem oder strukturiertem Hintergrund durchgeführt und anschließend unter Verwendung der Tiefensegmentierung wiederholt. Die Resultate sind in Tabelle 6.4 zusammengefasst.

Testumgebung	Testobjekt					
	Flasche	Hammer	Kopfhörer	Becher	Schere	Teddy
Keine Segmentierung (weißer Hintergrund)	✓	✓	✓	✗	✓	✓
Keine Segmentierung (strukt. Hintergrund)	✗	✓	✓	✗	✗	✓
Tiefensegmentierung (weißer Hintergrund)	✓	✓	✓	✓	✓	✓
Tiefensegmentierung (strukt. Hintergrund)	✓	✓	✓	✓	✓	✓

Tabelle 6.4: Ergebnisse der Klassifikation mit verschiedenen Hintergründen mit und ohne Tiefensegmentierung

Dabei wird deutlich, dass die Klassifikation der Klassen Flasche, Becher und Schere stark vom Szenenhintergrund abhängig sind. Das Objekt Becher kann sogar unabhängig von der Hintergrundstruktur nicht klassifiziert werden. Unter Anwendung der Tiefensegmentierung hingegen können alle Testobjekte unabhängig vom Hintergrund klassifiziert werden. Bild 6.15 verdeutlicht die Vorteile der Tiefensegmentierung am Beispiel der Sonderfälle [Becher, Schere]. Somit konnte durch die Verwendung von Tiefeninformationen die Erkennung oder Klassifikation der Objekte deutlich verbessert werden.

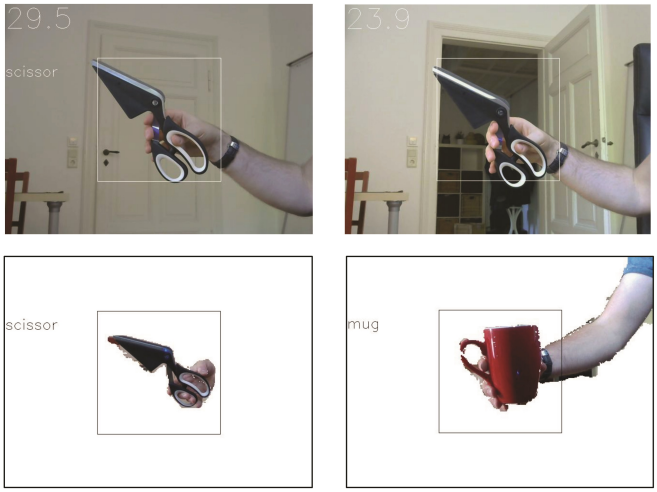

Abbildung 6.15: Verbesserung der Klassifikation durch Tiefensegmentierung am Beispiel der Sonderfälle [Becher, Schere]

Gestützt durch diese Ergebnisse soll für die Umsetzung der Klassifikation eine Tiefensegmentierung der Eingangsbilder als Vorverarbeitungsschritt vorgesehen werden.

Kapitel 7

Hardware- und Systemdesign

7.1 AeonCam Stereokamera-Plattform

Als Zielplattform zur Umsetzung von FPGA-basierten, binären Neuronalen Netzen wurde als vorhergehende Forschung die Stereokamera-Plattform AeonCam entwickelt (siehe Bild 7.1). Die AeonCam Plattform wurde bereits unter [MMNB17] veröffentlicht.

7.1.1 Heterogenes Systemdesign der AeonCam Plattform

Die Basis der Plattform bilden ein Xilinx Zynq XC7Z030 FPGA in Verbindung mit einer USB Type C Schnittstelle und einer speziell entwickelten Stereokamera-Einheit. Die Besonderheit dieses Designs ist der modulare Aufbau, bei dem der FPGA Chip auf die jeweilige Anwendung in Größe und Performanz angepasst werden kann. So können neben dem Xilinx Zynq XC7Z030 FPGA auch die Varianten XC7Z015, XC7Z020, XC7Z045 verbaut werden. Das Design mit dem Zynq XC7Z030 FPGA-Modul bietet einen eingebetteten Kintex 7 FPGA-Teil mit 125k Logikelementen, 78.600 LUTs, 156.200 FlipFlops und einen BlockRAM Speicher von 9.3Mb in Verbindung

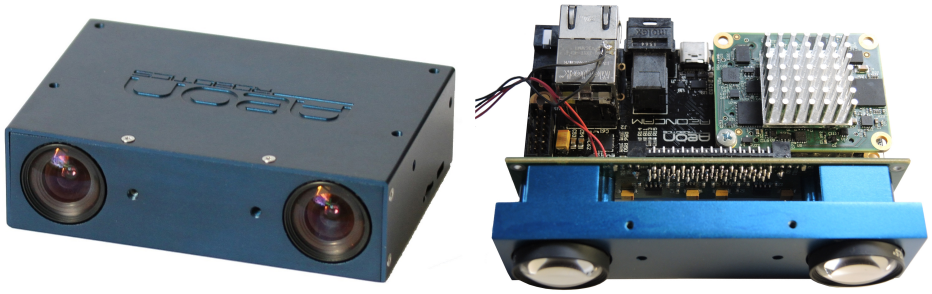

Abbildung 7.1: Stereokamera-Plattform AeonCam

mit einem ARM Dual Core Prozessor mit einer 667 MHz Prozessortaktung. Eine Übersicht des Systemdesigns ist in Abbildung 7.2 gegeben.

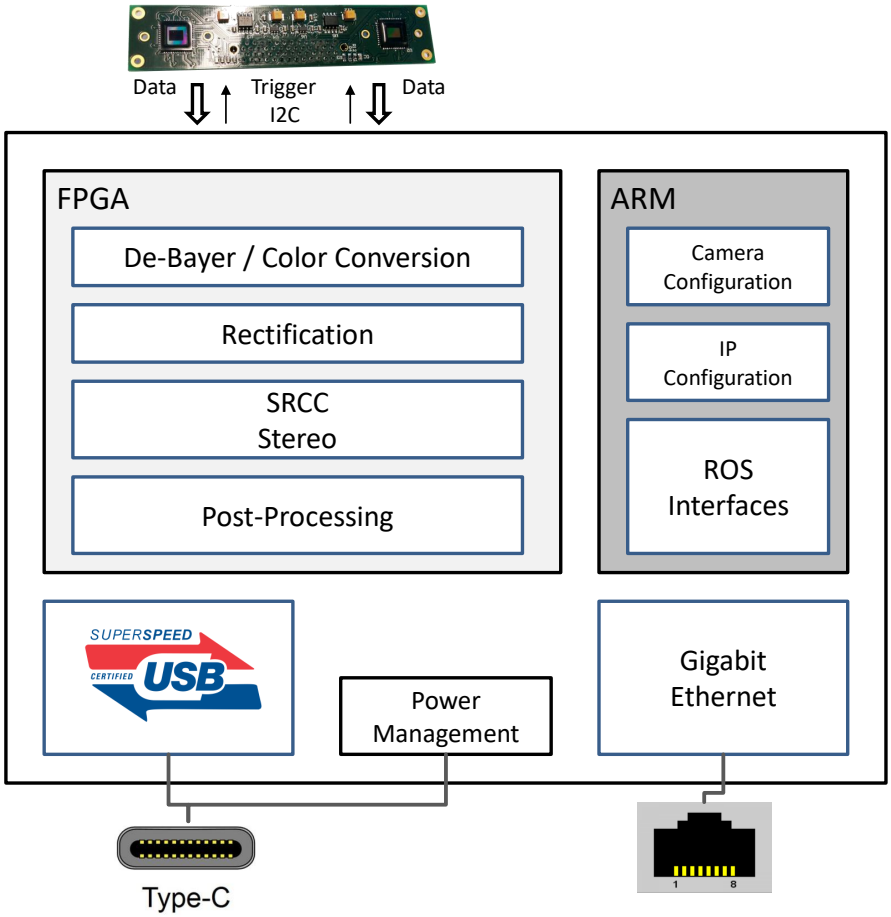

Abbildung 7.2: Systemdesign der AeonCam Plattform

Die aufwändigen Bildverarbeitungsalgorithmen zur Berechnung von Tiefeninformationen aus stereoskopischen Bildern sind als Hardware-Implementierung im FPGA-Teil umgesetzt. Die Module zur Konfiguration der Kamerasensoren und der Hardware-IP-Blöcke sowie ein ROS Kommunikationsinterface werden als Software auf dem ARM Prozessor ausgeführt. Es konnte in dieser Veröffentlichung die komplexe Berechnung von Tiefeninformationen durch verschiedene Module zur Stereoverarbeitung einschließlich Vor- und Nachverarbeitung mit einer Geschwindigkeit von 60 Bildern pro Sekunde umgesetzt werden. Das Kameradesign zeichnet sich zusätzlich durch seine geringe Größe von 110mm x 75mm x 31mm und den geringen Strom-

verbrauch von unter 5W aus. Da die Stereoverarbeitung dieser Arbeit zur Tiefensegmentierung verwendet werden soll, werden die zugehörigen Verarbeitungselemente im folgenden Abschnitt genauer erläutert.

7.1.2 Elektrisches Design der AeonCam Plattform

Die AeonCam Plattform basiert auf einem komplexen PCB-Design, welches die zwei Kamerasensoren über ein paralleles Interface mit 12 Datenleitungen und vier Steuerungsverbindungen pro Kamerasensor mit dem FPGA-Modul zusammenbringt. Dabei sind die Leitungslängen aufeinander abgestimmt, um eine Übertragungsverzögerung zwischen den Sensoren zu verhindern. Zusätzlich wird die Bildaufnahme beider Sensoren über ein Trigger-Signal synchronisiert. Die Hochgeschwindigkeitsübertragung der Ausgangsdaten wurde über 32 parallele Datenleitungen über einen Cypress FX3 Baustein zu einer zum UVC Standard kompatiblen SuperSpeed USB 3.0 / USB Type C Verbindung konvertiert. Dabei wurden die differentiellen Ausgangsleitungen der USB und PCIe Interfaces gemäß der Spezifikationen mit einem differentiellen Aufbau mit einer Impedanz von 90 Ohm herausgeführt. Dafür wurden die Leitungsbreiten und der Lagenaufbau der Platine entsprechend abgestimmt.

7.1.3 Mechanisches Design der AeonCam Plattform

Das mechanische Design der AeonCam Plattform wurde mit einem CAD-Tool entworfen und über ein CNC-Verarbeitungsverfahren aus Aluminium mit einer A6061 Legierung gefertigt. Das Design besteht insgesamt aus 3 Komponenten - aus einem speziellen Kameramodul, das die M12 Kamera-Objektive miteinander verbindet und damit das optische Stereo-System stabilisiert, einem Grundaufbau der die Platine fixiert und die Aussparungen für die Anschlüsse umfasst, sowie einen Verschlussdeckel, der das Kamerasystem verschließt. Diese Komponenten werden miteinander durch Schraubverbindungen zu einem stabilen Gehäuse zusammengeführt. Im Anschluss des Fertigungsprozesses wurden die mechanischen Komponenten mit einer Eloxierung in RAL5001 Optik bearbeitet, um die entsprechende Färbung zu erreichen und das Material zu härten.

7.1.4 Stereo Verarbeitung

Stereo-Algorithmen generieren Tiefeninformationen zu Bildszenen, in dem sie Korrespondenzen zwischen zwei nebeneinander angeordneten Eingangsbildern bilden. Dabei kann generell zwischen lokalen und globalen Methoden zur Bestimmung der Korrespondenzen unterschieden werden. In der Forschung zu der vorgestellten Arbeit wurde hierfür ein spezieller SRCC Algorithmus “Sparse Retina Census Correlation” als Kombination aus zwei

lokalen Methoden entwickelt: Durch eine Kombination der “Sum of Absolute Differences” (SAD) und dem “Sum of Hamming Distances” (SHD) Algorithmus konnte ein robuster und effizienter Algorithmus zur Berechnung von Tiefeninformationen erarbeitet werden. Bild 7.3 gibt eine schematische Darstellung der Verarbeitung.

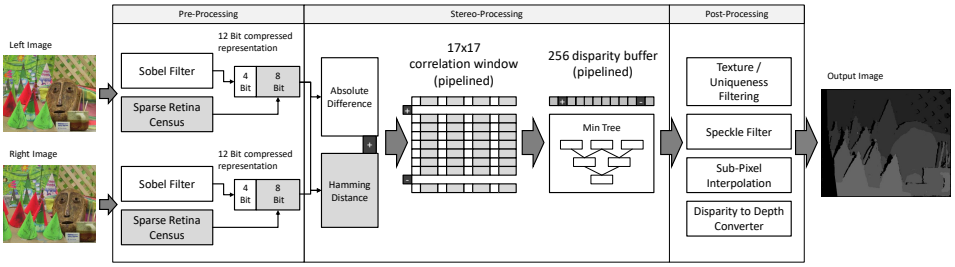

Abbildung 7.3: SRCC Stereo Verarbeitungskette

Die Eingangsbilder werden als Vorverarbeitungsschritt getrennt über Fensterbasierte Sobel- und Census-Transformationen für beide Algorithmen aufbereitet. Dabei wurde für die Census-Transformation ein eigenes Pattern aus [FA13] zusammengestellt. Das Pattern entspricht einer Auswahl von 8 markanten Punkten nach einer Retina-ähnlichen Verteilung. Die 8-Bit Repräsentationen der 5x5 Fenster der Census-Transformation werden mit den 4-Bit Informationen des 5x5 Sobel-Filters zusammengeführt und als komprimierte 12-Bit Darstellung an den nächsten Verarbeitungsschritt weitergereicht. Es werden die Korrelationskosten als Kombination des “Sum of Absolute Differences” (SAD) Algorithmus

$$C_{SAD} = \sum_{(x,y)\in W} |I_R(x,y) - I_L(x+d,y)| \tag{7.1}$$

und des “Sum of Hamming Distances” (SHD) Algorithmus

$$C_{SHD} = \sum_{(x,y)\in W} Hamming(C_R(x,y), C_L(x+d,y)) \tag{7.2}$$

durchgeführt.
Hierfür werden die Informationen der auf 12-Bit komprimierten Darstellung über ein Akkumulator-Modul in einem 17x17 Pixel großen Fenster aufsummiert. Entsprechend dieser Berechnung werden die Korrespondenzen des beweglichen Fensters des linken Eingangsbildes mit 256 weiteren Fenstern des rechten Eingangsbildes erstellt. Dadurch ergeben sich 256 Korrespondenzwerte, die über Maximierung die beste Übereinstimmung in Form der Disparität darstellen. Zusätzlich wird das zweitbeste Ergebnis verwendet,

um einen “uniqueness Filter” und eine subpixel Interpolation zu implementieren. Eine Textur-Filterung bildet eine Summe der lokalen Strukturen der Fenster, um Ergebnisse ohne aussagekräftige Strukturen auszusortieren. Des Weiteren wird ein sogenannter “Speckle”-Filter angewendet, um herausstechende Bildsegmente des Tiefenbilds entsprechend nach Bild 7.4 zu entfernen.

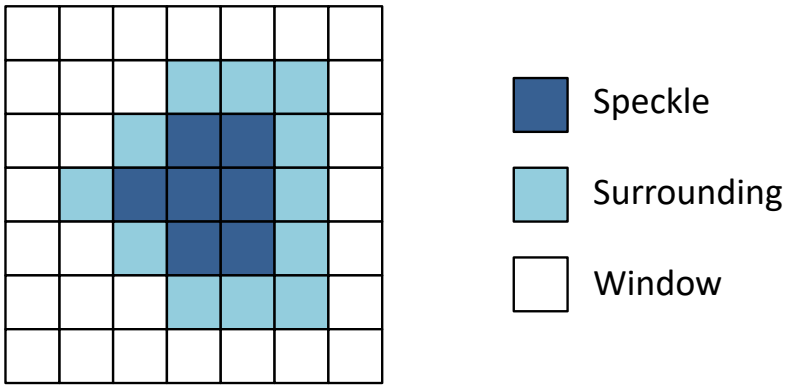

Abbildung 7.4: Speckle filter

7.1.5 Vorverarbeitung

De-Bayer/Farbkonvertierung

Zur Aufbereitung der Rohdaten der Bildsensoren mit Bayer-Filter-Format müssen die Pixelwerte zu Farbinformationen für jeden Bildpunkt umgewandelt werden. Diese Umformung ist in Darstellung 7.5 gezeigt.
So ermittelt das implementierte Debayer-Modul über ein 3x3 Fenster die RGB und YUV Farbinformationen für jeden Eingangswert. Dabei benötigt das Modul lediglich 7 Takte, um die Umformung durchzuführen.

Stereo-Rektifizierung

Im Zuge der Arbeit wurde ebenfalls eine OpenCV kompatible Implementierung einer Rektifizierung umgesetzt. Eine Rektifizierung der Eingangsbilder ist für die weitere Stereoverarbeitung unerlässlich, um die Bildinformationen entlang der Epipolarlinien auszurichten und somit eine Disparität ermitteln zu können. Hierzu benutzt das erstellte Hardwaremodul die Kameramatrix C und die Rotations- und Translationsmatrix RT

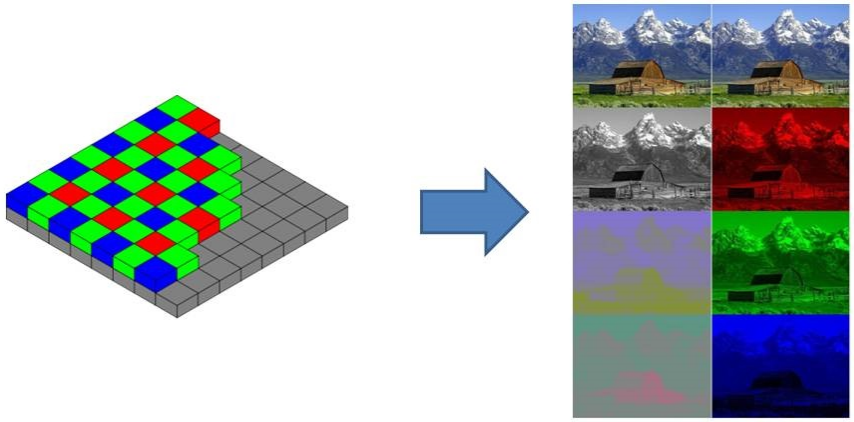

Abbildung 7.5: Farbkonvertierung der Eingangsdaten in die RGB und YUV Farbräume

$$C = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \qquad RT = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \tag{7.3}$$

zusammen mit der Verzerrungsmatrix D

$$D = (k_1, k_2, k_3, p_1, p_2) \tag{7.4}$$

als Eingangsparameter für die Umformung der Bilder. Diese Parameter werden im Vorfeld durch eine Kalibrierung ermittelt. Das Modul ermittelt die rektifizierten Pixelkoordinaten nach folgender Berechnung:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R * \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} + t \tag{7.5}$$

$$u = f_x x_d + c_x \tag{7.6}$$
$$v = f_y y_d + c_y \tag{7.7}$$

$$x_d = x'(1 + k_1 r^2 + k_2 r^4) \tag{7.8}$$
$$y_d = y'(1 + k_1 r^2 + k_2 r^4) \tag{7.9}$$

$$x' = x/z \qquad y' = y/z \tag{7.10}$$

$$r^2 = x'^2 + y'^2 \tag{7.11}$$

So konnte eine Echtzeit-Rektifizierung der Eingangsbilder mit einer sehr geringen Latenz von nur 40 Takten umgesetzt werden. Das Ergebnis des Rektifizierungsmoduls ist in Abbildung 7.6 dargestellt. Die gezeigte Hilfslinie soll die Ausrichtung der beiden Bilder zueinander verdeutlichen.

Abbildung 7.6: Ergebnis der Rektifizierung der beiden Eingangsbilder in Graustufen

7.1.6 Ergebnisse der AeonCam Plattform

Die implementierte Stereoverarbeitungspipeline mit eingeschlossener Vor- und Nachverarbeitung unter Verwendung der SRCC-Stereoverarbeitung berechnet hochpräzise 720p Tiefenbilder in einer Bildrate von 60 Bildern pro Sekunde. Die Ergebnisbilder dieser Verarbeitung sind in Bild 7.7 präsentiert.

Abbildung 7.7: Ergebnisse der AeonCam Plattform in unterschiedlichen Umgebungen - Links: Rektifiziertes Farbbild, Rechts: Berechnetes Tiefenbild

Die Performanz des Systems wurde besonders im Vergleich mit anderen State-of-the-Art Stereoimplementierungen durch Einführung eines MDE/s Wertes (Million Disparity Estimation per second) deutlich.

$$MDE/s = image\ resolution * disparities * fps \tag{7.12}$$

Reference	MDE/s (10^6)	fps	Algorithm	Platform	nonocc	all	disc
Proposed	14156	60	SRCC	FPGA	5.75	6.24	NA
Zha et al. [ZJX16]	5806	30	cross tree	FPGA	6.96	12.1	17.7
Werner et al [WSR14]	5806	30	NCC	FPGA	NA	16.3	NA
Humenberger et al [HZW+10]	1152	75	Census	GPU	7.96	13.8	20.3
Yoon et al. [YK06]	17.2	7	SAD	CPU	7.88	13.3	18.6

Tabelle 7.1: Vergleich der MDE/s Werte und der Fehlerraten (Middlebury Stereo Dataset [SHK+14], Teddy, bad 1.0)

So ist der erreichte MDE/s Wert um den Faktor 2,4x höher als die bisher beste FPGA Umsetzung und 12,3x schneller als die beste GPU Implementierung. Im Vergleich zur schnellsten CPU Variante ist die vorgestellte Implementierung sogar 823x schneller. Dabei ist das Design hinsichtlich der benötigten FPGA Ressourcen, wie in Tabelle 7.2 aufgeführt, vergleichsweise sparsam. Dies zeigt auch der geringe Energieverbrauch von unter 5W.

Resource	Stereo Pipeline, 128 Disp.	Stereo Pipeline, 256 Disp.	Available
LUT	32665 (41,6%)	47970 (61,0%)	78600
LUTRAM	4417 (16,6%)	6910 (26,0%)	26600
FlipFlops	51487 (32,8%)	76319 (48,6%)	157200
BlockRAM	156.5 (59,1%)	156.5 (59,1%)	265
DSP	57 (14,3%)	57 (14,3%)	400

Tabelle 7.2: FPGA Ressourcenverbrauch des Zynq XC7Z030 SoC

Aufgrund der guten Performanz und der bereits in Hardware implementierten Tiefenberechnung sollen die verbleibenden Logikressourcen und der bestehende ARM Prozessor zur Umsetzung der Klassifikation mittels eines Binären Neuronalen Netzes verwendet werden. Im Kapitel 7.2 wird die Umsetzung von Neuronalen Netzen auf dem Xilinx Zynq FPGA beschrieben. Für die Implementierung wurde dafür das größere Xilinx XC7Z045 FPGA eingesetzt, um mehr Logikressourcen bereitzustellen.

7.2 FPGA-basierte Umsetzung des Binären Neuronalen Netzes

Als Grundstruktur des Hardware-basierten Binären Neuronalen Netzes wird die Implementierung aus [ZSZ+17] verwendet. Im Kapitel 6.3 wurde bereits eine ideale Netzarchitektur mit einem 9-lagigen Aufbau mit einer Eingangsauflösung von 64x64 Pixeln bestimmt.

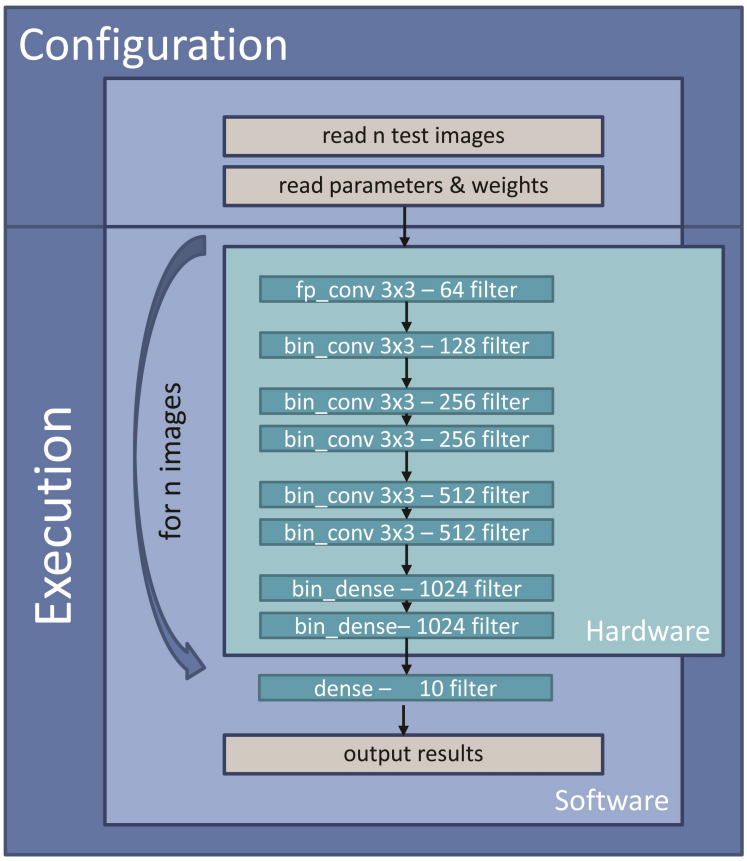

Abbildung 7.8: Ablauf der Hardware-basierten Klassifikation

Um diese Architektur auf dem FPGA umzusetzen, muss dementsprechend der Lagenaufbau neu implementiert werden. Außerdem muss die Eingangsauflösung der Basisimplementierung von 32x32 Pixel auf 64x64 erweitert werden. Entsprechend werden über die Eingangsdaten die Speicherarchitektur und im Besonderen die AXI-Bus-Kommunikation zwischen ARM-Prozessor und den Hardwaremodulen beeinflusst. Die Hardwareimplementierung verwendet die trainierten Modellparameter und Gewichte in einer komprimierten [W,K,H] Parameter-Darstellung nach

$$k = \frac{\gamma}{\sqrt{\sigma^2 + \epsilon}} \qquad h = \beta - \frac{\mu\gamma}{\sqrt{\sigma^2 + \epsilon}} \tag{7.13}$$

Deshalb wurde das AL BNN Framework entsprechend erweitert, um die Parameter (KH) und Gewichte (W) am Ende des Trainingsprozesses in diese Darstellung umzuformen und als "al_training_WKH_params.zip" für die spätere Verwendung abzuspeichern. Zusätzlich werden die Testdaten zusammen mit den zugehörigen Label-Referenzen unter "al_test_dataset.zip" und "al_dataset_labels.zip" abgelegt. Diese Datenarchive werden eingelesen und entsprechend dem in Bild 7.8 dargestellten Ablauf-Diagramm verarbeitet.
Dabei handelt es sich um ein heterogenes Systemdesign, in dem modulweise eine Hardware-Software Partitionierung vorgenommen wurde. So wird das Einlesen der Daten und die Konfiguration des Neuronalen Netzes in Software implementiert. Die Ausführung des Neuronalen Netzes ist über Hardware Module realisiert. Zur Kommunikation zwischen beiden Partitionen wird der AXI-Systembus eingebunden. Diese Designmethodik ist in Kapitel 7.2.5 detaillierter beschrieben.
Die Netzarchitektur ist in der Konfigurationsdatei AccelTest.h über die Parameter

```
unsigned N_LAYERS = 9;
unsigned L_CONV = 6;
unsigned S_tab[] =  { 64,  32,  16,  16,   8,   8,    4,    1,    1};
unsigned M_tab[] =  {  3,  64, 128, 256, 256, 512, 8192, 1024, 1024};
unsigned N_tab[] =  { 64, 128, 256, 256, 512, 512, 1024, 1024,   10};
unsigned T_tab[] =  {  0,   1,   1,   1,   1,   1,    2,    2,    3};
unsigned widx_tab[] = {0,   3,   6,   9,  12,  15,   18,   21,   24};
unsigned kidx_tab[] = {1,   4,   7,  10,  13,  16,   19,   22,   25};
unsigned hidx_tab[] = {2,   5,   8,  11,  14,  17,   20,   23,   26};
unsigned pool_tab[] = {1,   1,   0,   1,   0,   1,    0,    0,    0};
```

abgelegt. Dabei entspricht N_LAYERS der Gesamtlagenanzahl und L_CONV der Anzahl der Faltungslagen. Über S_tab[], M_tab und N_tab werden für jede Lage die Eingangsgröße, die Kanalanzahl und die Kernanzahl vorgegeben. Der T_tab bestimmt nach

```
0:= Gleitkommazahl Faltungslage
1:= Bin{"a}re Faltungslage
2:= Dense (Fully-Connected)-Lage
```

den Funktionstyp der Lage.
Die Zuweisungsreihenfolge der Gewichte und der KH Parameter aus den Archivdateien ist über die widx_tab, kidx_tab und hidx_tab Angaben definiert. Der pool_tab gibt entsprechend für jede Lage an, ob am Ende ein Pooling vorgenommen werden soll. Die Software Komponente des Systems liest in der Konfigurationsphase zunächst eine vorgegebene Anzahl an Testbilder

mit Label-Referenzen ein. Die farbigen Pixeldaten der Testbilder sind hier als 20-Bit Werte pro Farbkanal in quantisierter Darstellung vorhanden und sind in 64 Bit Wörtern ohne Überlappung repräsentiert. Dies ist in Grafik 7.9 verdeutlicht.

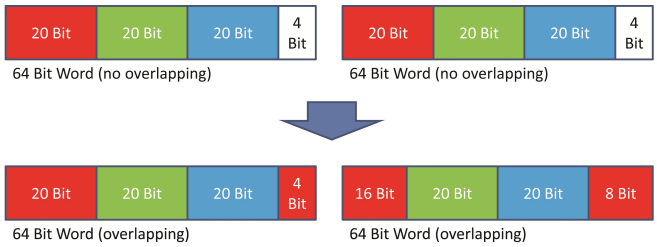

Abbildung 7.9: Datenstruktur der Farbbilder in 64 Bit Wortgröße mit und ohne Überlappung

Ein Testbild umfasst 64x64 Pixel und hat somit eine Gesamtgröße von 4096 Wörtern. Zur Konfiguration der Netzarchitektur werden anschließend die KH Parameter und die Gewichte eingelesen. Über die Funktion compute_accel_schedule wird nun die Architektur entsprechend der beschriebenen Konfiguration lagenweise erstellt. Gleichzeitig werden in diesem Schritt die zugehörigen Gewichte und Parameter übergeben. Zur Ausführung des Binären Neuronalen Netzes wird über run_accel_schedule ein Scheduling-Verfahren implementiert, das die Lagen entsprechend der vorherigen Konfiguration in Reihenfolge ausführt. Dafür werden die Eingangsdaten jeder Lage entsprechend der Lagenkonfiguration übergeben und über die Hardwaremodule “fp_conv”, “bin_conv” und “bin_dense” entsprechend prozessiert. Die Ein-und Ausgänge der Daten sind für jedes Hardwaremodul einheitlich als

```
void top(
    Word wt_i[WT_WORDS],
    Word kh_i[KH_WORDS],
    Word dmem_i[DMEM_WORDS],
    Word dmem_o[DMEM_O_WORDS],
    const Address    n_inputs,
    const Address    n_outputs,
    const Address    input_words,
    const Address    output_words,
    const ap_uint<3> layer_mode,  // [0]='new layer', [2:1]='conv1,
        conv,dense'
    const ap_uint<1> dmem_mode,   // 0 means dmem[0] is input
    const ap_uint<2> width_mode,  // 0=8'b, 1=16'b, 2=32'b, 3=64'b
    const ap_uint<2> norm_mode    // 0='do nothing', 1='do norm', 2='
        do pool'
);
```

in der Struktur des “top” Moduls implementiert. So können die jeweiligen

Module zur Laufzeit eingebunden werden und können auf gemeinsamen Datenstrukturen arbeiten. Die Funktionsweise der einzelnen Hardware-Module wird im folgenden Abschnitt erläutert.

7.2.1 Floating-Point Convolutional Layer

Das fp_conv Modul wird als erste Lage der Netzarchitektur verwendet, um die Eingangsbilder in Gleitkommazahl-Darstellung anzunehmen. Die einzelnen Testbilder liegen hier in der Struktur von 64x64 Bildpunkten mit 3 Farbkanälen [64x64x3] vor. Die ist entsprechend im S_tab[] mit 64 und im M_tab[] mit 3 vorgegeben. Nach der ausgewählten Netzarchitektur soll hier eine Faltung über 64 Faltungskerne der Größe 3x3 vorgenommen werden. Hierfür wurde das fp_conv Hardwaremodul so implementiert, dass es ein 3x3 Fenster aufbaut und über die 4096 Eingangswörter eine Faltung nach 2.1 durch Hardware parallelisiert berechnet. Die Anzahl der parallel-arbeitenden Modulinstanzen wird über "CONVOLVERS" definiert. Durch die binären Gewichte kann das Ergebnis der Faltung "res" wie folgt bestimmt werden:

```
res += (b==0) ? pix : (C1InputType)(-pix);
```

Hier wird abhängig vom binären Gewicht b addiert oder subtrahiert. Dies entspricht der Formel

$$y_n = f(\sum_{m=1}^{M} x_m * w_{n,m} + b_n) \tag{7.14}$$

aus dem Grundlagen-Kapitel mit Gewichten $w_n = [1, -1]$ und dem Bias $b_n = 0$. Durch die Anwendung von 64 Faltungskernen auf der Bildfläche mit 64x64 Pixeln entstehen die Ausgangsergebnisse ($4096x64BitWörter$) für jede Kernanwendung.
Die Besonderheit der binären Implementierung ist die nachfolgende Normierung. Hier wird über

```
outwords_tmp[(r-1)/2][((r-1)%2)*S + (c-1)] = (res >= nc) ? Bit(0) :
    Bit(1);
```

das Ergebnis der Faltung "res" mit dem aus KH vorgegebenen Normierungsparameter "nc" verglichen und entsprechend einem binären Wert [Bit(0), Bit[1]] zugeordnet. Die Ergebnisse werden also in eine binäre Darstellung zu 4096 Bit umgeformt. Dieses Zwischenergebnis wird als Struktur

```
DWord outwords_tmp[32]
```

angelegt, wobei DWord einer Größe von 128 Bit (2xWortgröße) entspricht. Diese spezielle Darstellung ist für das anschließende Pooling wichtig. Ein Vorteil der gewählten Netzarchitektur ist, dass neben der Normierung auch das Pooling-Verfahren direkt in der ersten Lage angewendet wird, um die Daten für die nachfolgenden Berechnungen deutlich zu reduzieren. Da die Daten bereits in binärer Form abgelegt sind, kann das 2x2 Pooling durch eine logische "UND"-Verknüpfung ($\wedge$) in horizontaler und vertikaler Richtung umgesetzt werden. Diese Berechnung ist besonders für FPGA-basierte Systeme sehr effizient zu realisieren. Durch das Pooling reduziert sich die Ausgangstruktur auf

```
Word outwords[16]
```

So werden abschließend nur 16 Wörter in binärer 64 Bit Darstellung (16x64 = 1024 Bit), beziehungsweise 32x32 Bit für jeden der 64 Kerne an die nachfolgenden Ebenen des Netzes weitergereicht. Die nachfolgende Lage hat dementsprechend ein Eingangsvolumen von 32x32x64 Bit.

7.2.2 Binärer Convolutional Layer

Das binäre Faltungsmodul entspricht der Implementierung aus [ZSZ+17] und führt durch die gegebene Netzkonfiguration Faltungsberechnungen mit 128, 256 oder 512 Kernen der Größe 3x3 aus. Um die Ergebnisse wieder im binären Format darzustellen, folgt hier eine Batch-Normalisierung. Wie in der Netzarchitektur vorgegeben, folgt nach jeder zweiten Faltungslage eine 2x2 Pooling-Umformung. Durch die Reihe an Faltungs-und Pooling-Lagen reduziert sich das Ausgangsvolumen nach der letzten binären Faltungslage auf 4x4x512=8192 Bit.

7.2.3 Binärer Dense Layer

Der Dense Layer führt, wie in Kapitel 2.1 bereits beschrieben, ein Punktprodukt zwischen den trainierten Gewichten und den Eingangsdaten durch. In einem Binären Netz entspricht diese Berechnung einer "XOR"-Verknüpfung der Eingangsdaten mit den trainierten Gewichten und einer anschließenden Aufsummierung der gesetzten Bits. Dies korrespondiert somit mit der Bildung der Hamming Distanz. Um die abschließende Binarisierung vorzunehmen, wird das Ergebnis wie bei den Faltungslagen mit einem Normierungsparameter nc verglichen und nach

```
o_word[o_offset] = (sum >= nc) ? 0 : 1;
```

einem binären Wert zugeordnet. Das Ausgangsvolumen beträgt nach dem letzten binären Dense Layer 1x1x1024 Bit.

7.2.4 Software Dense Layer

Die letzte Lage des Binären Neuronalen Netzes wurde als Software Funktion "last_layer_cpu" aus [ZSZ+17] implementiert. Dadurch kann die Anzahl der möglichen klassifizierbaren Objekte von 1 bis zur maximalen Anzahl "num_classes" konfiguriert werden, ohne das die Hardware über die im nachfolgenden Kapitel vorgestellte High-Level-Synthese neu generiert werden muss. Außerdem soll das Ausgangsergebnis in nicht binärer Darstellung berechnet werden, um alle Objektklassen abbilden zu können. Dadurch ändert sich die abschließende Berechnung des Software Dense Layers. Die Eingangsdaten werden über die Bildung des Punktprodukts mit den trainierten Gewichten und anschließender Aufsummierung verarbeitet. Der Normierungsparameter val wird allerdings nun als Gleitkomma-Zahl über eine Multiplikation mit dem k Parameter und der Addition mit dem h Parameter nach

```
float val = static_cast<float>(sum) * k_data+ h_data;
```

bestimmt. Durch Iteration über die Klassenanzahl [NUM_CLASSES] mit der Zählvariable n und die Abfrage:

```
if (pred == -1 || val > maxval) {
      pred = n;
      maxval = val;
    }
```

wird eine nicht binäre Aussage über die Klasse als Maximierung des Wertes val als Ergebnis der Klassifikation bestimmt.

7.2.5 Xilinx SDx High-Level-Design Methodik

Die SDx Entwicklungsumgebung der Firma Xilinx [Xil18b] ermöglicht die Implementierung von Hardware-Software-Co Designs für verschiedene Xilinx FPGAs. Dabei wird die SDSoC Umgebung für die Entwicklung heterogener Systeme basierend auf einem ARM Prozessor und einem dedizierten FPGA-Element angewendet. Durch diese Methodik kann eine Hardware-Software-Partitionierung während des Entwicklungsprozesses vorgenommen werden, um die begrenzte Rechenleistung des eingebetteten ARM Prozessors durch das angebundene FPGA-Element um parallel-agierende Hardwarekomponenten zu erweitern. Dabei wird, wie in Darstellung 7.10 gezeigt, mit einer C oder C++ Applikation begonnen und diese um High-Level-Synthese Befehle (typedef, Pragmas) erweitert.

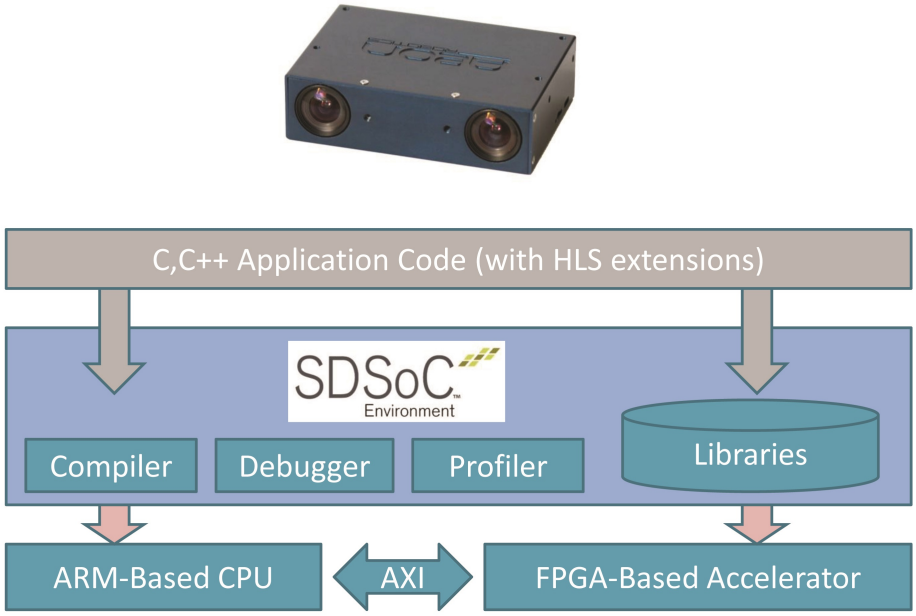

Abbildung 7.10: SDx High-Level-Design

Der auf Eclipse basierende Designfluss setzt die Softwarekomponenten über Kompilierung zu einer ausführbaren Software für den ARM Prozessor um. Gleichzeitig werden aus den Hardwarepartitionen über den Xilinx HLS Designfluss unter Anwendung bestehender Bibliotheken Hardwaremodule generiert. Die Kommunikation zwischen den Hardwarekomponenten und der Software ist über den AXI Systembus realisiert. Um die Datenübertragung entsprechend umsetzen zu können und die Speichereinheiten für Daten zu erstellen, müssen im HLS Designfluss die benötigten Variablen mit Bitbreiten definiert werden. Der folgende Ausschnitt gibt ein Beispiel für diese Definitionen.

```
typedef ap_int<WORD_SIZE> Word;
typedef ap_int<WORD_SIZE*2> DWord;
typedef ap_int<WT_SIZE> WtType;
typedef ap_uint<16> Address;
typedef ap_int<12> ConvSum;
typedef ap_int<5> ConvOut;
typedef ap_uint<10> IdxType;
typedef ap_fixed<16,4> C1Comp;
typedef ap_int<16> NormComp;
typedef ap_int<16> DenseSum;
typedef ap_fixed<16,12> DenseNorm;
typedef ap_fixed<20,2, AP_RND> C1InputType;
typedef ap_fixed<24,6, AP_RND> C1ConvType;
```

Parallele Verarbeitung von Schleifen

In einem C/C++ Programm werden Befehle innerhalb einer Schleife normalerweise sequenziell ausgeführt. Es wird also der nächste Durchlauf einer Schleife erst nach Beendigung der vorhergehenden Berechnungen angestoßen. In der Xilinx High-Level-Synthese hingegen gibt es die Möglichkeit über Angabe von Pragmas die Ausführung von Schleifen zu parallelisieren. Hierzu können die Pragmas HLS PIPELINE und HLS UNROLL angewendet werden.
Ein Beispiel für die HLS PIPELINE Anweisung ist hier gegeben:

```
for (index_a = 0; index_a < A_NROWS; index_a++) {
     for (index_b = 0; index_b < B_NCOLS; index_b++) {
#pragma HLS PIPELINE II=1
          float result = 0;
          for (index_d = 0; index_d < A_NCOLS; index_d++) {
              float product_term = in_A[index_a][index_d] * in_B[
                  index_d][index_b];
              result += product_term;
          }
          out_C[index_a * B_NCOLS + index_b] = result;
     }
}
```

In dieser Implementierung ist der Parameter “Initiation Interval” (II) auf einen Takt gesetzt. Hier wird die nächste Schleifeniteration direkt nach dem Lesebefehl (RD) ausgeführt. Dadurch werden die Operationen nach Bild 7.11 in einer Pipeline-Struktur bearbeitet.

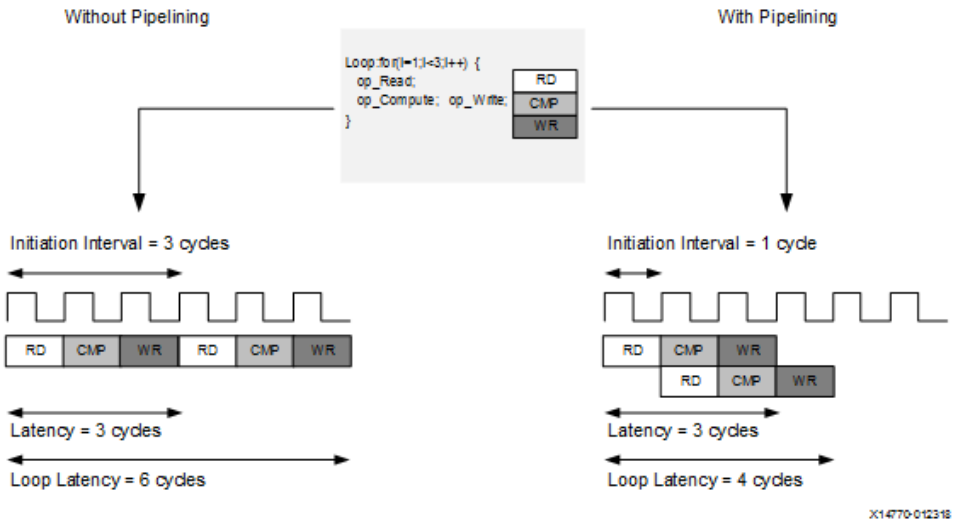

Abbildung 7.11: Programmablauf mit PIPELINE Befehl, Quelle [Xil18a]

Ein weiteres Verfahren, um Schleifen zu parallelisieren, ist durch den Parameter HLS UNROLL gegeben. Hier werden für jeden iterativen Durchlauf einer Schleife Kopien erstellt, die dadurch vollständig parallel ausgeführt werden können. Der folgende Programmausschnitt zeigt einen normalen Schleifenaufruf ohne Parallelisierung:

```
int sum = 0;
for(int i = 0; i < 10; i++) {
    sum += a[i];
}
```

Nach Anwendung der Unroll-Technik mit einem Faktor von 2 entsteht:

```
int sum = 0;
for(int i = 0; i < 10; i+=2) {
    sum += a[i];
    sum += a[i+1];
}
```

So kann in diesem Beispiel die Schleife mit maximal 10 Kopien parallel bearbeitet werden.
In der Implementierung aus [ZSZ+17] wurde das Hardwaremodule "bin_conv" durch Definition von Schleifen zur Aufteilung der Verarbeitung in Phasen erweitert. Hier wird eine bestimmte Anzahl von Daten ("WORDS_PER _PHASE") über die parallelen Einheiten "CONVOLVERS" verarbeitet. Dabei wurde der Befehl HLS UNROLL angewendet, um die spezielle parallele Ausführung zu erreichen. Auch in den Modulen "fp_conv" und "bin_dense" wurden Schleifen zur Parallelisierung eingeführt. Die Strukturen zur Beschleunigung durch Parallelisierung einzelner Lagen aus [ZSZ+17] wurden auch zur Umsetzung in dieser Arbeit angewendet.

Kapitel 8

Ergebnisse

In dieser Arbeit wurde ein neuartiges Framework zum Trainieren von Neuronalen Netzen zur Objektklassifikation entwickelt. Es wurde ein Prozess zur automatisierten Generierung von Datensätzen eingeführt, der auf Basis von Suchmaschinen Bildmaterial automatisiert zusammenträgt und für das Training einer Bildklassifikation aufbereitet.

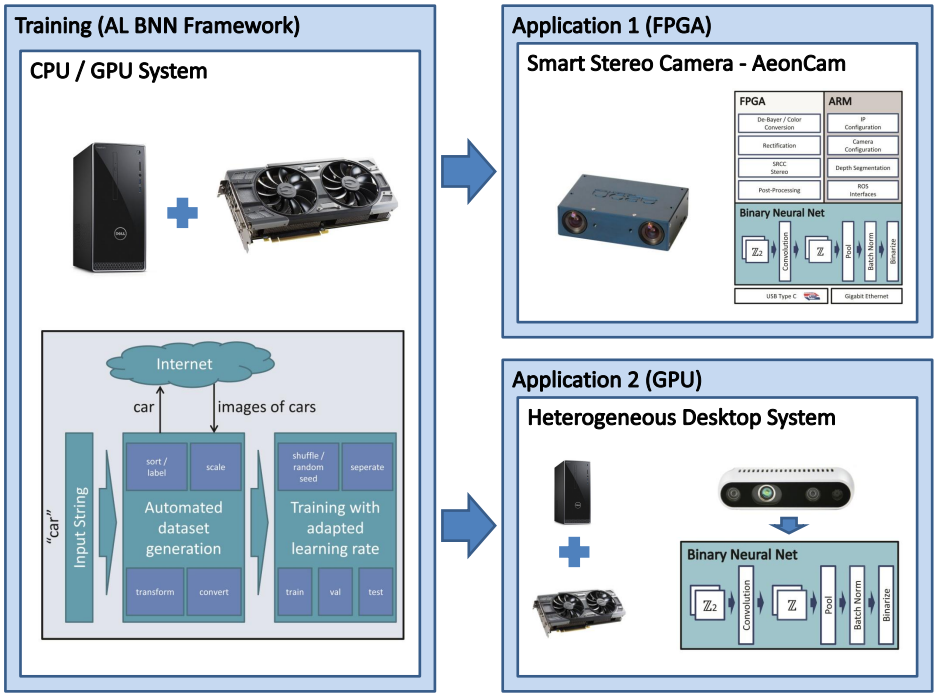

Abbildung 8.1: Übersicht des heterogenen Systemdesigns des AL BNN Frameworks

Die dadurch reduzierte Datensatzgröße wurde durch Anpassung des Trainingsprozesses kompensiert. Außerdem wurde eine quantisierte Netzarchitektur angewendet, um die Gewichte und Daten des Netzes durch eine Binarisierung weiter zu reduzieren. Das vorgestellte AL BNN Framework liefert nach dem automatisierten Trainingsprozess Modelle für unterschiedliche heterogene Systeme. Es wurde die FPGA-basierte Stereokamera-Plattform AeonCam entwickelt, um die Klassifizierung auf Hardware-unterstützten, eingebetteten Systemen zu demonstrieren. Die erstellten Modelle können zusätzlich auf einem GPU-basierten System eingesetzt werden. Das entwickelte Systemdesign für den automatisierten Trainingsprozess und der anschließenden Ausführungsphase ist in Bild 8.1 gezeigt.

8.1 Verbesserung der Klassifikation

Unter Verwendung des AL BNN Frameworks konnte die Netzarchitektur speziell auf die Gegebenheiten der automatisch-generierten Datensätze angepasst werden. So konnte eine optimale Eingangsauflösung von $64x64$ Pixeln der Eingangsbilder bestimmt werden und die Netzarchitektur aus Abbildung 8.2 als idealer Aufbau für Datensätze mit reduzierter Größe gefunden werden.

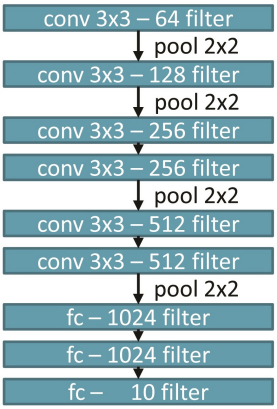

Abbildung 8.2: Finale Netzarchitektur des AL BNN Frameworks

Die Architektur bietet besonders durch die kompakte Struktur in Anlehnung an die VGG-11/16 Netzarchitektur und dem vereinfachten Lagenaufbau ausgezeichnete Performanz bei kleinen Datensätzen. Dieses Verhalten konnte durch die finalen Trainingsergebnisse des AL BNN Frameworks mit einer Klassifikationsgenauigkeit von 82,3% nachgewiesen werden. Die Evaluation der Verwendung von Tiefeninformationen aus Kapitel 6.4 hat ergeben, dass die Tiefeninformationen besonders zur Umsetzung einer Tiefensegmentierung der Eingangsbilder geeignet sind. Hier konnte über eine

Testreihe mit und ohne Tiefensegmentierung, wie in Tabelle 8.1 dargestellt ist, der Einfluss auf die Klassifikation gezeigt werden. Durch den Einsatz der Tiefensegmentierung konnten die Klassifikationsergebnisse unabhängig vom Szenenhintergrund gestaltet und weiter verbessert werden.

Testumgebung	Testobjekt					
	Flasche	Hammer	Kopfhörer	Becher	Schere	Teddy
Keine Segmentierung (weißer Hintergrund)	✓	✓	✓	✗	✓	✓
Keine Segmentierung (strukt. Hintergrund)	✗	✓	✓	✗	✗	✓
Tiefensegmentierung (weißer Hintergrund)	✓	✓	✓	✓	✓	✓
Tiefensegmentierung (strukt. Hintergrund)	✓	✓	✓	✓	✓	✓

Tabelle 8.1: Ergebnisse der Klassifikation mit verschiedenen Hintergründen mit und ohne Tiefensegmentierung

8.2 Trainingsergebnisse des AL BNN Frameworks

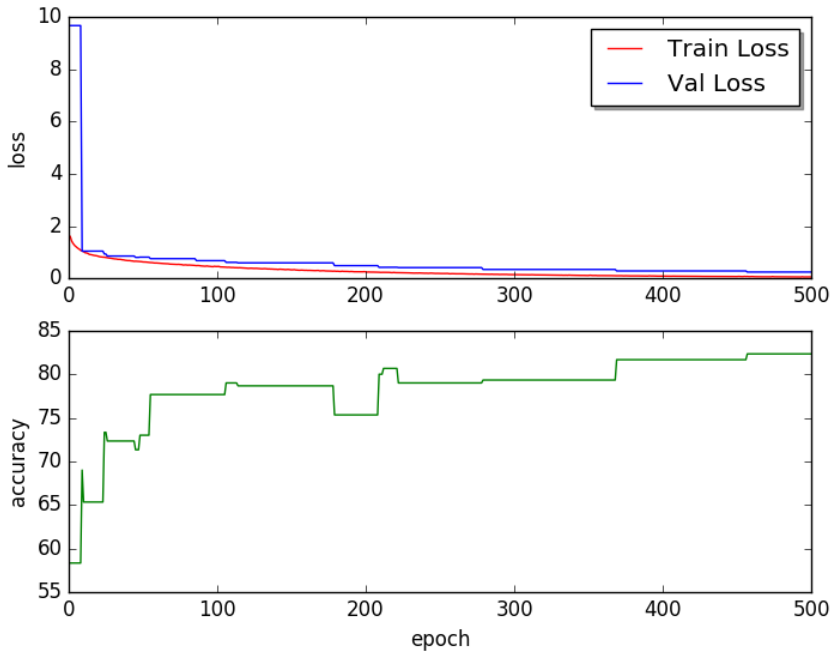

Abbildung 8.3: Trainingsverlauf mit dem AL BNN Framework als Darstellung über Klassifikationsgenauigkeit und Trainingsverlust

Durch die direkte Verbindung der automatisierten Generierung mit dem eingesetzten Trainingsframework "Theano" konnte eine ideale Abstimmung der Trainingsparameter auf Datensatz und Netzarchitektur gefunden werden. Der Trainingsverlauf mit diesen Einstellungen ist grafisch in 8.3 dargestellt. Die zugehörige Konfusionsmatrix ist in Bild 8.4 aufgeführt.

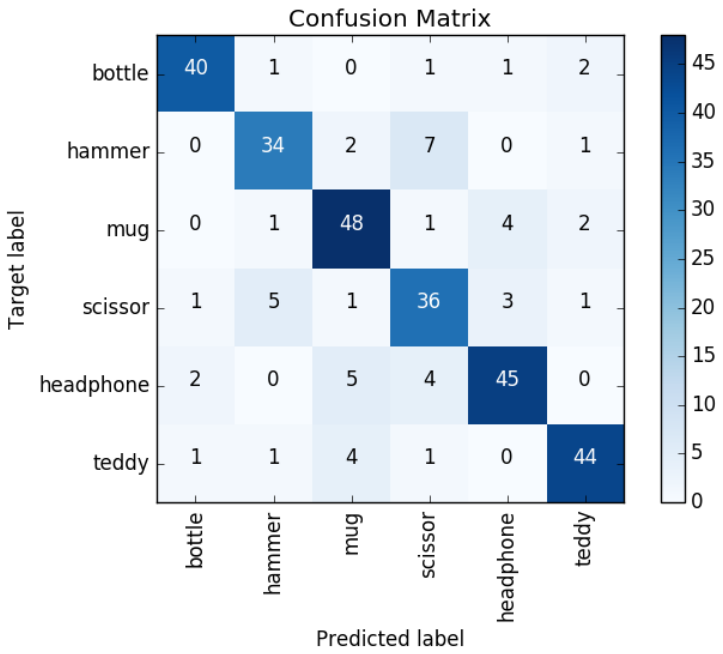

Abbildung 8.4: Trainingsergebnis in Darstellung der Konfusionsmatrix

In dieser Darstellung wird deutlich, dass die Klassifizierungsergebnisse der Objektklassen "mug", "headphones" und "teddy" mit über 88% sogar deutlich höher ausfallen als das Gesamtergebnis des Trainings. Die Klassifikationen der Objekte "hammer" und "scissor" sind hingegen oft vertauscht und sind somit schwierig voneinander zu unterscheiden.
Das Trainingsergebnis im Vergleich zum Training des CIFAR-10 Datensatzes ist in Tabelle 8.2 aufgeführt.

	CIFAR10	AL BNN
Genauigkeit der Klassifikation	88,2%	82,3%

Tabelle 8.2: Ergebnis der Klassifikation im Vergleich zum CIFAR-10 Training

Trotz der starken Reduktion der Datensatzgröße kann das AL BNN Framework mit einer lediglich um 6% verringerten Klassifikationsgenauigkeit mit dem State-of-the-Art Training des CIFAR-10 Datensatzes konkurrieren. Die Datensatzgröße wurde von 60.000 Bildern auf 1800 Bilder und somit um den Faktor 33,3x komprimiert.

8.2.1 Vergleich der Trainingszeiten

Ein explizites Ziel dieser Arbeit war die Beschleunigung des Trainingsprozesses. Um die Vorteile des AL BNN Frameworks in quantisierbaren Zahlen darzustellen, wurde die in Kapitel 6 aufgestellte Auflistung der Teilschritte mit dem jeweiligen Zeit- oder Ressourcenaufwand für einen Vergleich herangezogen. Der Vergleich dieser Werte mit den Ergebnissen des AL BNN Frameworks ist in Tabelle 8.3 präsentiert.

	CIFAR10		AL BNN	
Vorgang	Zeitaufwand	Ressourcen	Zeitaufwand	Ressourcen
Erstellen des Datensatzes*	~ 1-2 Wochen	~ 0,5 PM	23 Minuten	4,8Wh
Architekturdesign & Trainingsparameter	~ 1-2 Wochen	~ 0,5 PM	---	---
Training (CPU + GPU)**	15 Stunden	2587,5Wh	33 Minuten	94,9Wh
Gesamt	2-4 Wochen	1 PM + 2587,5Wh	56 Minuten	99,7Wh

* Intel Core i7 6700 @ 3,4 GHz : Leistungsverbrauch 12,5W

** Intel CPU + GTX 970 GPU : Intel CPU + GTX 970 GPU : Leistungsverbrauch 21,5W + 151W = 172,5W

Tabelle 8.3: Vergleich der Trainingszeiten

Die im Vorfeld als aufwändigsten Teilschritte bestimmten Prozesse zur Erstellung des Datensatzes und Design der Netzarchitektur mit den zugehörigen Anpassungen der Trainingsparametern konnten durch die Umsetzung des Automatisierten Lernens von 2-4 Wochen um den Faktor 1000x auf 23 Minuten reduziert werden. Die Konsolenausgaben dieses Prozessschrittes sind hier angegeben:

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 250.454898119s
--- created class hammer in 185.149206161s
--- created class mug in 206.077868938s
--- created class scissor in 243.146188021s
--- created class headphone in 266.492260933s
--- created class teddy in 208.866183996s
Dataset generation took: 0h 22m 40s
```

Die zugehörigen Konsolen-Ausgaben für das Training sind zusammen mit der Zeitmessung und dem Trainingsverlauf nachfolgend aufgeführt.

```
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
Training...
--- Epoch 1 of 500 took 5.04864382744s
--- test loss:           9.66779613494873
--- test accuracy rate:  58.33333383003871%
--- time left 0h 42m 4s
--- Epoch 50 of 500 took 3.6572599411s
--- test loss:           0.8113660911719004
--- test accuracy rate:  73.00000016887982%
--- time left 0h 27m 29s
--- Epoch 100 of 500 took 3.72813796997s
--- test loss:           0.677117109298706
--- test accuracy rate:  77.66666673123837%
--- time left 0h 24m 54s
--- Epoch 150 of 500 took 3.71836400032s
--- test loss:           0.5926511685053507
--- test accuracy rate:  78.66666701932749%
--- time left 0h 21m 45s
--- Epoch 200 of 500 took 3.70509195328s
--- test loss:           0.4864436239004135
--- test accuracy rate:  75.33333326379457%
--- time left 0h 18m 35s
--- Epoch 250 of 500 took 3.74805998802s
--- test loss:           0.41137540837128955
--- test accuracy rate:  79.00000015894571%
--- time left 0h 15m 40s
--- Epoch 300 of 500 took 3.74528479576s
--- test loss:           0.340203399459521
--- test accuracy rate:  79.33333329856396%
--- time left 0h 12m 32s
--- Epoch 350 of 500 took 3.73108100891s
--- test loss:           0.340203399459521
--- test accuracy rate:  79.33333329856396%
--- time left 0h 9m 23s
--- Epoch 400 of 500 took 3.72829389572s
--- test loss:           0.28338484714428586
--- test accuracy rate:  81.66666639347872%
--- time left 0h 6m 16s
--- Epoch 450 of 500 took 3.76073694229s
--- test loss:           0.28338484714428586
--- test accuracy rate:  81.66666639347872%
--- time left 0h 3m 11s
--- Epoch 500 of 500 took 3.76583099365s
--- test loss:           0.2410653680562973
--- test accuracy rate:  82.3333335419496%
--- time left 0h 0m 3s
Final Accuracy: 82.3333335419496
Training took: 0h 33m 4s
```

Durch die Verwendung einer binären Netzimplementierung in Kombination mit der reduzierten Datensatzgröße konnte auch der Trainingsprozess beschleunigt werden. So konnten die anfangs festgestellten 15 Stunden für das Training einer Objektklassifikation ohne Änderung der Testsystem-Hardware um den Faktor 27x auf 33 Minuten gesenkt werden.
Über das Linux Tool "turbostat" konnte die Leistungsaufnahme des Testrechners mit einem verbauten Intel Core i7 6700 Prozessor von 12,5W bei Erstellung des Datensatzes bestimmt werden. Daraus folgt ein Ressourcenverbrauch von 4,8Wh für die automatisierte Generierung des Trainingsdatensatzes. Für den Trainingsprozess mit GPU Unterstützung konnte ein Verbrauch von insgesamt 94,9Wh gemessen werden. Hierfür wurden sowohl die Daten des "turbostat" Tools zur Bestimmung des CPU Verbrauchs als auch die Angabe des Nvidia Tools "nvidia-smi" zur Bestimmung des GPU Verbrauchs ausgewertet.
Zusammenfassend kann durch das AL BNN Framework der gesamte Trainingsprozess einschließlich der Erstellung eines passenden Datensatzes in einer Zeit von unter einer Stunde ausgeführt werden. Für den Gesamtleistungsverbrauch konnte ein Verbrauch von 99,7Wh ermittelt werden.

8.3 Ressourcenverbrauch des FPGA-basierten heterogenen Systems

Das FPGA-basierte heterogene System, das in Kapitel 7 vorgestellt wurde, lässt sich über die Anzahl der parallelen Recheneinheiten und der Tiefenstufen (Disparities) der Stereo Verarbeitung skalieren. Die benötigten FPGA Ressourcen für das AL BNN Hardwaredesign mit unterschiedlicher Anzahl paralleler Einheiten sind in Tabelle 8.4 zusammenstellt. Die Angaben entsprechen der Implementierung auf einem Zynq XC7Z045 FPGA.

Resource	AL BNN, 2 Conv*	AL BNN, 4 Conv*	AL BNN, 8 Conv*	Available
LUT	43333 (19,8%)	49412 (22,6%)	56077 (25,7%)	218600
LUTRAM	622 (0,88%)	622 (0,88%)	626 (0,89%)	70400
FlipFlops	26662 (12,2%)	29424 (13,5%)	28920 (13,2%)	218600
BlockRAM	89 (16,3%)	91 (16,7%)	106 (19,5%)	545
DSP	4 (0,44%)	4 (0,44%)	4 (0,44%)	900

* Anzahl der parallelen Hardwareeinheiten ("Convolvers")

Tabelle 8.4: Ressourcenverbrauch des AL BNN Frameworks auf dem Zynq XC7Z045 SoC

Durch die Anpassung der parallelen Einheiten können die benötigten Ressourcen von beispielsweise 19,8% der Logikzellen (LUT) bis 25,7% der verfügbaren Logik-Ressourcen konfiguriert werden. Dabei ist zu beachten, dass die Performanz des Systems abhängig von der Anzahl der parallelen Einheiten

ist. Eine detaillierte Analyse der Performanz der verschiedenen Konfigurationen ist im nachfolgenden Abschnitt 8.4 ausgeführt.
Die Tabelle 8.5 zeigt den Ressourcenverbrauch des AL BNN Frameworks in Kombination mit der SRCC Stereoverarbeitungspipeline auf dem Zynq XC7Z045 FPGA. In dieser Konfiguration wurde mit 8 parallelen Einheiten für das Hardwaredesign des AL BNN Frameworks gearbeitet und die Stereoverarbeitung mit 256 Tiefenstufen eingebunden.

Resource	**AL BNN, 8 Conv* + Stereo Pipeline, 256 Disp.**	**Available**
LUT	104047 (47,6%)	218600
LUTRAM	7536 (10,7%)	70400
FlipFlops	105239 (48,1%)	218600
BlockRAM	263 (48,3%)	545
DSP	61 (6,8%)	900

Tabelle 8.5: Ressourcenverbrauch des AL BNN Frameworks zusammen mit der SRCC Stereoverarbeitung auf dem Zynq XC7Z045 SoC

Durch den Verbrauch an FlipFlop und BlockRam Speicherelementen von 48,1% und 48,3% ist der FPGA zu ca. 50% ausgelastet. Die FPGA-basierte AeonCam Plattform bietet ein modulares Design. So kann neben dem Zynq XC7Z045 FPGA auch das kleinere und kostengünstigere Zynq XC7Z030 Modul eingebunden werden. Tabelle 8.6 zeigt den Ressourcenverbrauch dieses FPGAs bei Verwendung des AL BNN Designs mit Stereoverarbeitung zur Tiefensegmentierung. Damit das Design mit 96,7% Logikzellenverbrauch in das XC7Z030 Modul platziert werden kann, muss die Anzahl der parallelen Hardwareeinheiten auf zwei reduziert werden und die Tiefenstufen auf 128 Disparities verringert werden.

Resource	**AL BNN, 2 Conv* + Stereo Pipeline, 128 Disp.**	**Available**
LUT	75998 (96,7%)	78600
LUTRAM	5039 (18,9%)	26600
FlipFlops	78149 (49,7%)	157200
BlockRAM	246 (92,8,1%)	265
DSP	61 (15,2%)	400

Tabelle 8.6: Ressourcenverbrauch des AL BNN Frameworks zusammen mit der SRCC Stereoverarbeitung auf dem Zynq XC7Z030 SoC

8.4 Ausführungszeiten auf verschiedenen Rechenplattformen

Um die Ausführungszeiten des GPU-unterstützen Systems und des FPGA-basierten AeonCam Designs zu bestimmen, wurden die entsprechenden Implementierungen auf zwei unterschiedlichen Testsystemen ausgeführt. Das heterogene Desktop-System verwendet eine Nvidia GTX 970 GPU mit 4Gb internem Speicher in Verbindung mit einer Intel Core i7 6700 CPU mit 32 Gb Arbeitsspeicher und entspricht somit der Darstellung aus 8.1(Application 2). Für die Messung der Ausführungszeit des mit dem AL BNN Framework trainierten Modells wurde die Bildaufnahme und Tiefensegmentierung mit dem Intel Realsense D435 Kamerasensor realisiert und die Laufzeit der Klassifikation eines 64x64 Pixel großen Bildausschnittes gemessen. Die Leistungsaufnahme wurde wieder über die Tools “turbostat” und “nvidia-smi” bestimmt. Die erhobenen Daten sind in Tabelle 8.7 gezeigt.

Plattform	Ausführungszeit 64x64	Leistungs-aufnahme	Bilder/s pro Watt
Heterogenes Desktop System (GPU)*	3,65ms (274 fps**)	121,9W	2,248
AeonCam (FPGA, 2 Convolvers ||)	13,6ms (73 fps**)	4,6W	15,870
AeonCam (FPGA, 4 Convolvers ||)	10,7ms (94 fps**)	4,6W	20,435
AeonCam (FPGA, 8 Convolvers ||)	9,27ms (108 fps**)	4,7W	22,942

* Intel CPU + GTX 970 GPU : Leistungsverbrauch 21,9W + 100W = 121,9W

** fps = Bilder pro Sekunde

|| Anzahl der parallelen Hardwareeinheiten (“Convolvers”)

Tabelle 8.7: Ausführungszeiten auf der AeonCam Plattform und dem heterogenen Desktop-System

Die Performanz der Hardware-unterstützten Implementierung wurde auf dem Xilinx SoC ZC706 Testboard auf Basis des Xilinx XC7Z045 FPGAs aus Bild 8.5 evaluiert.
Hierfür wurde das in Kapitel 7 vorgestellte Hardwaredesign mit 8 parallelen Recheneinheiten konfiguriert und synthetisiert. Zur Laufzeitmessung wurde der Testdatensatz von der SD-Karte eingelesen und über das trainierte Modell des AL BNN Frameworks als Hardwareimplementierung klassifiziert. Die gemessene Ausführungszeit ist ebenfalls in Tabelle 8.7 aufgeführt. Die Bestimmung der Leistungsaufnahme wurde hierbei über das “Xilinx Power Estimation” Tool (XPE) vorgenommen. Die zugehörigen Konsolen-Ausgaben zur Klassifikation der 300 Bilder aus dem Testdatensatz sind hier aufgeführt:

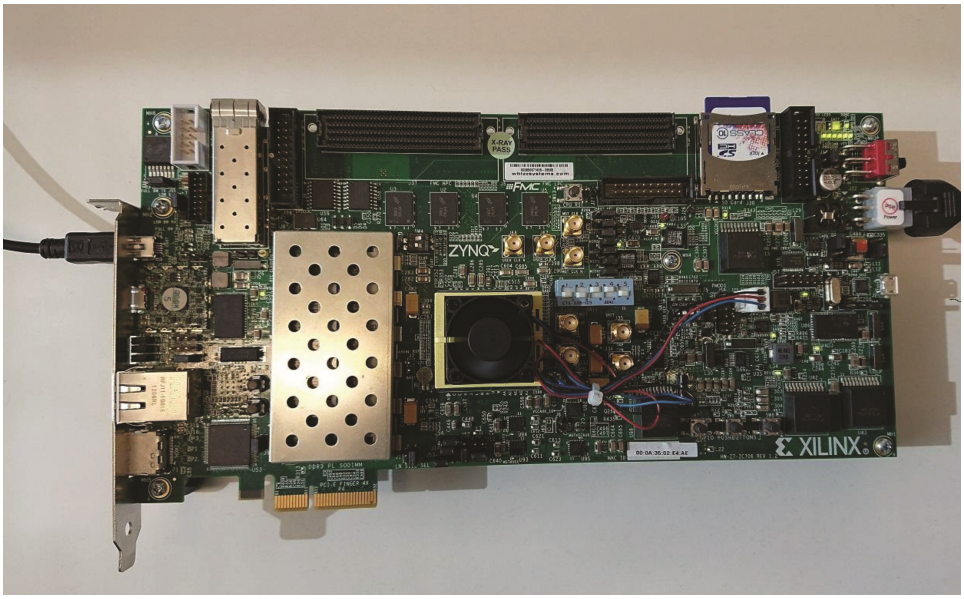

Abbildung 8.5: Testboard zur Messung der Laufzeit und Klassifikationsgenauigkeit auf Basis des Xilinx SoC ZC706 Entwicklungskits

```
False Prediction: 53 of 300 test images

BNN Accuracy: 82.33%

Total accel runtime =     2.7871 seconds

Total accel runtime =     2.7871 seconds

Runtime per Image =     0.0093 seconds

Images per sec =   107.6407

xl-Conv1             :    300 calls;  0.027 secs total time
xl-Conv2             :    300 calls;  1.626 secs total time
xl-Conv3             :    300 calls;  0.077 secs total time
xl-Conv4             :    600 calls;  0.168 secs total time
xl-Conv5             :   1200 calls;  0.176 secs total time
xl-Conv6             :   2400 calls;  0.207 secs total time
xl-FC1               :   9600 calls;  0.442 secs total time
xl-FC2               :   1200 calls;  0.065 secs total time
last                 :    300 calls;  0.003 secs total time
```

Neben der Gesamtlaufzeit zeigt sich, dass die im Training bestimmte Klassifikationsgenauigkeit von 82.3% auch unter Verwendung des FPGA-basierten Designs erreicht wird.

Zum Vergleich der beiden Systeme wurden die Daten in eine Darstellung der

Performanz in Bildern pro Sekunde und der Energieeffizienz in Bildern pro Sekunde pro Watt umgeformt. Beide Systeme bieten mit einer Bildwiederholrate von über 100 Bildern pro Sekunde eine ausgezeichnete Performanz. Bei Betrachtung der Bilder pro Sekunde pro Watt werden die Vorteile des FPGA-basierten AeonCam Systems deutlich. Hier liegt die Energieeffizienz des Gesamtsystems mit 22,94 Bildern pro Sekunde pro Watt deutlich über dem Wert des Desktop-Systems. Durch die Hardwareunterstützung konnte die Energieeffizienz um den Faktor 10,2x verbessert werden. Abschließend zeigt der Vergleich, dass beide Systeme spezifische Vorteile bieten. Das Desktop-System kann für Anwendungen, in denen sehr hohe Bildwiederholraten benötigt werden, zum Einsatz kommen. Hier ist zum Beispiel der Einsatz in Fahrer-Assistenz-Systemen, bei denen der eventuelle Bremsvorgang zeitnah eingeleitet werden muss, denkbar. Die eingebettete FPGA-basierte Implementierung auf der AeonCam Plattform ist unter der Betrachtung, dass viele Kamerasensoren nur eine maximale Bildaufnahmefrequenz von 60 Bildern pro Sekunde unterstützen und der deutlich besseren Energieeffizienz die passende Alternative für mobile Roboter oder Drohnen.

Kapitel 9

Zusammenfassung

Diese Arbeit hat sich intensiv mit der Thematik des Automatisierten Lernens von Neuronalen Netzen am Beispiel der Objektklassifikation auseinandergesetzt. Um den Lernprozess zu vereinfachen und zu beschleunigen, wurde die Idee der automatisierten Generierung von Datensätzen entwickelt und in Form eines eigenständigen Frameworks zum Automatisierten Lernen AL BNN umgesetzt. Der vorgestellte Ansatz ermöglicht aus Angaben von gewünschten Objektklassen über Suchmaschinen aus dem Internet Bildmaterialien zusammenzutragen und für den Trainingsprozess entsprechend aufzubereiten. Das AL BNN Framework bindet für den Trainingsprozess die bestehende Theano-Umgebung zum effizienten Trainieren Neuronaler Netze mit ein. Dadurch konnten die Trainingsparameter speziell auf die Gegebenheiten beim automatisierten Generieren von Datensätzen angepasst und das Trainingsergebnis somit deutlich verbessert werden. Für die Umsetzung des Trainings wurde ein heterogenes System auf GPU-Basis gewählt, da diese Systeme deutliche Vorteile in Bezug auf die Performanz und Flexibilität des Frameworks gezeigt haben. Um die Effizienz und Performanz der Ausführung der Objektklassifikation zu optimieren wurde hierfür ein Hardware-basiertes Systemdesign ausgewählt und erstellt. Durch Anwendung einer binären Quantisierung konnte die Hardware-Implementierung sehr effizient implementiert werden. Hierzu wurde das AL BNN Framework entsprechend um eine binäre Netzarchitektur erweitert.

Für die Ausführung der Objektklassifikation, wurde eigens für diese Arbeit ein eingebettetes System in Form der FPGA-basierten Stereokamera Plattform AeonCam entwickelt. Hierfür wurde über das elektrische und mechanische Design zusammen mit Umsetzung der Hardware und Software-Komponenten der komplette Prozess einer Produktentwicklung durchgeführt. Das so entstandene Design kann Tiefeninformationen über Hardware-Module in Echtzeit bereitstellen, um die Objektklassifikation durch Tiefensegmentierung deutlich zu verbessern. Dabei konnte die Performanz des Systems erfolgreich demonstriert werden. Die bestehende FPGA Implementierung

aus [ZSZ+17] wurde für die durch Evaluation gewählte Netzarchitektur und Eingangsauflösung angepasst und auf der FPGA-Testplattform umgesetzt und analysiert. Das AL BNN Framework konnte den gesamten Trainingsprozess einschließlich der Teilschritte zum Erstellen von eigenen Datensätzen von einem Zeitaufwand von mehreren Wochen auf 56 Minuten um den Faktor 1000x reduzieren. So kann das vorgestellte Framework durch den automatisierten Lernprozess ideal als Plattform zum Entwickeln eigener Netzarchitekturen und Anpassung der Trainingsparameter genutzt werden. Durch die einheitliche Struktur der generierten Datensätze kann es auch als Benchmark-Umgebung für bestehende Netzimplementierungen herangezogen werden. Bereits während der technischen Umsetzung in dieser Arbeit war das Framework zur Anpassung der eigenen Parameter ausgesprochen hilfreich.
Das AL BNN Framework erstellt aus gewünschten Objektklassen ein trainiertes Modell zur Klassifikation dieser Objekte mittels eines Binären Neuronalen Netzes. Dabei werden die Modellparameter direkt für die FPGA-basierte heterogene Zielplattform umgeformt und bereitgestellt. Über das trainierte Modell kann die FPGA-basierte Plattform die Objektklassifikation sehr energieeffizient ausführen. So konnte die Klassifikation mit einer Bildwiederholrate von 108 Bildern pro Sekunde auf dem eingebetteten System mit einem Energieverbrauch von lediglich 4,7W demonstriert werden. Die Energieeffizienz konnte somit um den Faktor 10,2x im Vergleich zum Desktop-System verbessert werden. Dadurch eignet sich das System besonders für den Einsatz in mobilen Robotern oder Drohnen. Zusätzlich kann das trainierte Modell auf einem GPU-basierten System ausgeführt werden, um die Bildwiederholrate auf Kosten der Energieeffizienz weiter zu erhöhen.

9.1 Ausblick

In dieser Arbeit wurde die deutlich verbesserte Effizienz der Ausführung Binärer Neuronaler Netze auf FPGA-basierten heterogenen Systemen gezeigt. In weiterführenden Arbeiten kann die Umsetzung des gesamten Trainingsprozesses auf eingebetteten Systemen untersucht werden. Dadurch werden neue Anwendungen für mobile Roboter denkbar. Es kann beispielsweise über die Umkehrung des Automatisierten Lernens über die Google Bilder-Rückwärtssuche zu einer gegebenen Szene ein bisher unbekanntes Objekt klassifiziert werden. Dabei wird entsprechend über eine Internetverbindung zu einem Objekt eine Klasse bestimmt. Durch das vorgestellte AL BNN Framework kann der mobile Roboter nun zu dieser Klasse Bildinformationen sammeln und das bis dahin unbekannte Objekt neu anlernen. Dadurch kann der Roboter nach erfolgreichem Abschluss des Trainingsprozesses das neue Objekt ohne bestehende Internetverbindung klassifizieren.

9.2 Danksagung

Ich möchte mich herzlich für die Unterstützung der beiden betreuenden Professoren Prof. Berekovic und Prof. Steil bedanken. Einen speziellen Dank möchte ich an Prof. Berekovic richten, der mich bei Fragen zur Umsetzung im Bereich des Chip Designs und mit dem großen Interesse zur Erschließung neuer Forschungsthemen unterstützt hat. Prof. Steil möchte ich für die Hilfe bei der strukturellen Umsetzung der Arbeit und das Einbringen seines weitreichenden Wissens im Bereich des maschinellen Lernens danken. Desweiteren möchte ich meiner Lebensgefährtin und meiner Familie danken, die mir dabei geholfen haben den Weg zur Promotion fokussiert zu verfolgen.

Abbildungsverzeichnis

Literaturverzeichnis

[ABC+16] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[ARAA+16] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, et al. Theano: A python framework for fast computation of mathematical expressions. *arXiv preprint*, 2016.

[BCN18] Léon Bottou, Frank E Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, 2018.

[Ber10] Prof. Dr.-Ing. Mladen Berekovic. Vorlesungsskript advanced vlsi design 2. Research report, C3E, Technische Universität Braunschweig, 2010.

[BHF+10] Christian Banz, Sebastian Hesselbarth, Holger Flatt, Holger Blume, and Peter Pirsch. Real-time stereo vision system using semi-global matching disparity estimation: Architecture and fpga-implementation. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*, pages 93–101. IEEE, 2010.

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg, 2006.

[BMSY11] Vincent Brost, Charles Meunier, Debyo Saptono, and Fan Yang. Flexible vliw processor based on fpga for real-time image processing. *Design and Architectures for Signal and Image Processing (DASIP)*, 2011.

[Bro17] Jason Brownlee. A gentle introduction to mini-batch gradient descent and how to configure batch size. Website, 2017. Available online at `https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/` visited on Aug 10th 2018.

[BTL13] Nadia Baha, Hakim Touzene, and Slimane Larabi. Fpga implementation for stereo matching algorithm. In *Science and Information Conference (SAI), 2013*, pages 448–454. IEEE, 2013.

[C+15] François Chollet et al. Keras: Deep learning library for theano and tensorflow. *URL: https://keras. io/k*, 7(8), 2015.

[CHS+16] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*, 2016.

[CTH+10] Nelson Yen-Chung Chang, Tsung-Hsien Tsai, Bo-Hsiung Hsu, Yi-Chun Chen, and Tian-Sheuan Chang. Algorithm and architecture of disparity estimation with mini-census adaptive support weight. *IEEE Transactions on Circuits and Systems for Video Technology*, 20(6):792–805, 2010.

[Den12] Li Deng. The mnist database of handwritten digit images for machine learning research [best of the web]. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.

[DHS11] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[Doz16] Timothy Dozat. Incorporating nesterov momentum into adam. 2016.

[FA13] Wade S Fife and James K Archibald. Improved census transforms for resource-optimized stereo vision. *IEEE Transactions on Circuits and Systems for Video Technology*, 23(1):60–73, 2013.

[GB10] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[Gsc16] David Gschwend. Zynqnet: An fpga-accelerated embedded convolutional neural network. *vol. Master ETH-Zurich: Swiss Federal Institute of Technology Zurich*, 2016.

[GWFM+13] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[He17] Yihui He. Estimated depth map helps image classification. *arXiv preprint arXiv:1709.07077*, 2017.

[Hin12] Geoffrey Hinton. Neural networks for machine learning, lecture 6a overview of mini-?batch gradient descent. Website, 2012. Available online at `http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf` visited on Aug 10th 2018.

[Hir08] Heiko Hirschmuller. Stereo processing by semiglobal matching and mutual information. *IEEE Transactions on pattern analysis and machine intelligence*, 30(2):328–341, 2008.

[HWLJ16] Jianhui Han, Zhen Wu, Lanying Li, and Ying Ji. Fpga implementation for binocular stereo matching algorithm based on sobel operator. *International Journal of Database Theory and Application*, 9(4):221–230, 2016.

[HZW+10] Martin Humenberger, Christian Zinner, Michael Weber, Wilfried Kubinger, and Markus Vincze. A fast stereo matching algorithm suitable for embedded real-time systems. *Computer Vision and Image Understanding*, 114(11):1180–1202, 2010.

[JSD+14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 675–678. ACM, 2014.

[Kar18] Andrej Karpathy. Convolutional neural networks for visual recognition. Website, 2018. Available online at `http://cs231n.github.io/convolutional-networks/` visited on Aug 10th 2018.

[Kat17] Vinod Kathail. Caffe to zynq: State-of-the-art machine learning inference performance in less than 5 watts. Website, 2017. Available online at `https://www.embedded-vision.com/sites/default/files/webinars/May%2024,%202017%20Webinar.pdf` visited on Aug 10th 2018.

[KB14] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[KFF15] Andrej Karpathy and Li Fei-Fei. Deep visual-semantic alignments for generating image descriptions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3128–3137, 2015.

[KPP12] Jedrzej Kowalczuk, Eric T Psota, and Lance C Perez. Real-time stereo matching on cuda using an iterative refinement method for adaptive support-weight correspondences. *IEEE transactions on circuits and systems for video technology*, 23(1):94–104, 2012.

[Kri17] Alex Krizhevsky. The cifar-10 dataset. Website, Ebook, 2017. Available online at `https://www.cs.toronto.edu/~kriz/cifar.html` visited on Aug 10th 2018.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[KZ01] Vladimir Kolmogorov and Ramin Zabih. Computing visual correspondence with occlusions using graph cuts. In *Computer Vision, 2001. ICCV 2001. Proceedings. Eighth IEEE International Conference on*, volume 2, pages 508–515. IEEE, 2001.

[LBRF11] Kevin Lai, Liefeng Bo, Xiaofeng Ren, and Dieter Fox. A large-scale hierarchical multi-view rgb-d object dataset. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 1817–1824. IEEE, 2011.

[LCY13] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[LGT16] Chen-Yu Lee, Patrick W Gallagher, and Zhuowen Tu. Generalizing pooling functions in convolutional neural networks: Mixed, gated, and tree. In *Artificial Intelligence and Statistics*, pages 464–472, 2016.

[LJY17] Fei-Fei Li, Justin Johnson, and Serena Yeun. Convolutional neural networks for visual recognition, lecture 9: Cnn architectures. Website, 2017. Available online at `http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture9.pdf` visited on Aug 10th 2018.

[LSL15] Fayao Liu, Chunhua Shen, and Guosheng Lin. Deep convolutional neural fields for depth estimation from a single image. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5162–5170, 2015.

[LTA16] Darryl Lin, Sachin Talathi, and Sreekanth Annapureddy. Fixed point quantization of deep convolutional networks. In *International Conference on Machine Learning*, pages 2849–2858, 2016.

[LZCS14] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM, 2014.

[MGB] Xiaobai Ma, Zhenglin Geng, and Zhi Bie. Depth estimation from single image using cnn-residual network.

[MMNB17] Soenke Michalik, Soeren Michalik, Jamin Naghmouchi, and Mladen Berekovic. Real-time smart stereo camera based on fpga-soc. In *Humanoid Robotics (Humanoids), 2017 IEEE-RAS 17th International Conference on*, pages 311–317. IEEE, 2017.

[MP15] Stefano Mattoccia and Matteo Poggi. A passive rgbd sensor for accurate and real-time depth sensing self-contained into an fpga. In *Proceedings of the 9th International Conference on Distributed Smart Cameras*, pages 146–151. ACM, 2015.

[MSSS13] Matthias Michael, Jan Salmen, Johannes Stallkamp, and Marc Schlipsing. Real-time stereo vision: Optimizing semi-global matching. In *Intelligent Vehicles Symposium (IV), 2013 IEEE*, pages 1197–1202. IEEE, 2013.

[MSZ+11] Xing Mei, Xun Sun, Mingcai Zhou, Shaohui Jiao, Haitao Wang, and Xiaopeng Zhang. On building an accurate stereo matching system on graphics hardware. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 467–474. IEEE, 2011.

[Nes83] Yurii Nesterov. A method for unconstrained convex minimization problem with the rate of convergence o (1/kˆ 2). In *Doklady AN USSR*, volume 269, pages 543–547, 1983.

[NSS+16] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: comparison of fpga, cpu, gpu, and asic. In *Field-Programmable Technology (FPT), 2016 International Conference on*, pages 77–84. IEEE, 2016.

[Nvi08] CUDA Nvidia. Cublas library. *NVIDIA Corporation, Santa Clara, California*, 15(27):31, 2008.

[PPAGAE+16] M Pérez-Patricio, Abiel Aguilar-González, M Arias-Estrada, Héctor-Ricardo Hernandez-de Leon, Jorge-Luis Camas-Anzueto, and JA de Jesús Osuna-Coutiño. An fpga stereo matching unit based on fuzzy logic. *Microprocessors and Microsystems*, 42:87–99, 2016.

[Qia99] Ning Qian. On the momentum term in gradient descent learning algorithms. *Neural networks*, 12(1):145–151, 1999.

[QML15] Affaq Qamar, Fahad Bin Muslim, and Luciano Lavagno. Analysis and implementation of the semi-global matching 3d vision algorithm using code transformations and high-level synthesis. In *Vehicular Technology Conference (VTC Spring), 2015 IEEE 81st*, pages 1–5. IEEE, 2015.

[RDS+15] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, et al. Imagenet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, 2015.

[RKK18] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. 2018.

[Rud18] Sebastian Ruder. An overview of gradient descent optimization algorithms. Website, 2018. Available online at `http://ruder.io/optimizing-gradient-descent/index.html#adam` visited on Aug 10th 2018.

[SBD+16] Vaddi Chandra Sekhar, Satyajit Bora, Monalisa Das, Pavan Kumar Manchi, S Josephine, and Roy Paily. Design and implementation of blind assistance system using real time stereo vision algorithms. In *VLSI Design and 2016 15th International Conference on Embedded Systems (VLSID),*

2016 29th International Conference on, pages 421–426. IEEE, 2016.

[Sch15a] Konstantin Schauwecker. Sp1: Stereo vision in real time. In *MuSRobS@ IROS*, pages 40–41, 2015.

[Sch15b] J. Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[SH16] David Silver and Demis Hassabis. Alphago: Mastering the ancient game of go with machine learning. *Research Blog*, 2016.

[SHK+14] Daniel Scharstein, Heiko Hirschmüller, York Kitajima, Greg Krathwohl, Nera Nešić, Xi Wang, and Porter Westling. High-resolution stereo datasets with subpixel-accurate ground truth. In *German Conference on Pattern Recognition*, pages 31–42. Springer, 2014.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[TM17] Inc The MathWorks. Introducing deep learning with matlab. Website, Ebook, 2017. Available online at `https://de.mathworks.com/content/dam/mathworks/tag-team/Objects/d/80879v00_Deep_Learning_ebook.pdf` visited on Aug 10th 2018.

[TMDSA08] Federico Tombari, Stefano Mattoccia, Luigi Di Stefano, and Elisa Addimanda. Near real-time stereo based on effective cost aggregation. In *Pattern Recognition, 2008. ICPR 2008. 19th International Conference on*, pages 1–4. IEEE, 2008.

[TT14] Christos Ttofis and Theocharis Theocharides. High-quality real-time hardware stereo matching based on guided image filtering. In *Proceedings of the conference on Design, Automation & Test in Europe*, page 356. European Design and Automation Association, 2014.

[WGB04] John Iselin Woodfill, Gaile Gordon, and Ron Buck. Tyzx deepsea high speed stereo vision system. In *Computer Vision and Pattern Recognition Workshop, 2004. CVPRW'04. Conference on*, pages 41–41. IEEE, 2004.

[WSR14] Martin Werner, Benno Stabernack, and Christian Riechert. Hardware implementation of a full hd real-time disparity estimation algorithm. *IEEE transactions on consumer electronics*, 60(1):66–73, 2014.

[Xil18a] Inc. Xilinx. Loop pipelining and loop unrolling. Website, 2018. Available online at `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/calling-coding-guidelines/concept_pipelining_loop_unrolling.html` visited on Aug 10th 2018.

[Xil18b] Inc. Xilinx. Sdsoc environment user guide. Website, 2018. Available online at `https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_2/ug1027-sdsoc-user-guide.pdf` visited on Aug 10th 2018.

[YJL+14] Qingqing Yang, Pan Ji, Dongxiao Li, Shaojun Yao, and Ming Zhang. Fast stereo matching using adaptive guided filtering. *Image and Vision Computing*, 32(3):202–211, 2014.

[YK06] Kuk-Jin Yoon and In So Kweon. Adaptive support-weight approach for correspondence search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(4):650–656, 2006.

[Zei12] Matthew D Zeiler. Adadelta: an adaptive learning rate method. *arXiv preprint arXiv:1212.5701*, 2012.

[ZFL+16] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-cnn: An fpga-based framework for training convolutional neural networks. In *Application-specific Systems, Architectures and Processors (ASAP), 2016 IEEE 27th International Conference on*, pages 107–114. IEEE, 2016.

[ZHAK08] Christian Zinner, Martin Humenberger, Kristian Ambrosch, and Wilfried Kubinger. An optimized software-based implementation of a census-based stereo matching algorithm. In *International Symposium on Visual Computing*, pages 216–227. Springer, 2008.

[ZJX16] Daolu Zha, Xi Jin, and Tian Xiang. A real-time global stereo-matching on fpga. *Microprocessors and Microsystems*, 47:419–428, 2016.

[ZKB04] Christopher Zach, Konrad Karner, and Horst Bischof. Hierarchical disparity estimation with programmable 3d hardware. 2004.

[ZLS+15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.

[ZSZ+17] Ritchie Zhao, Weinan Song, Wentao Zhang, Tianwei Xing, Jeng-Hau Lin, Mani Srivastava, Rajesh Gupta, and Zhiru Zhang. Accelerating binarized convolutional neural networks with software-programmable fpgas. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 15–24. ACM, 2017.

[ZZC+11] Lu Zhang, Ke Zhang, Tian Sheuan Chang, Gauthier Lafruit, Georgi Krasimirov Kuzmanov, and Diederik Verkest. Real-time high-definition stereo matching on fpga. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 55–64. ACM, 2011.

Kapitel 10

Anhang

AL BNN Framework mit 32x32 Pixeln und 9-lagiger Netzarchitektur

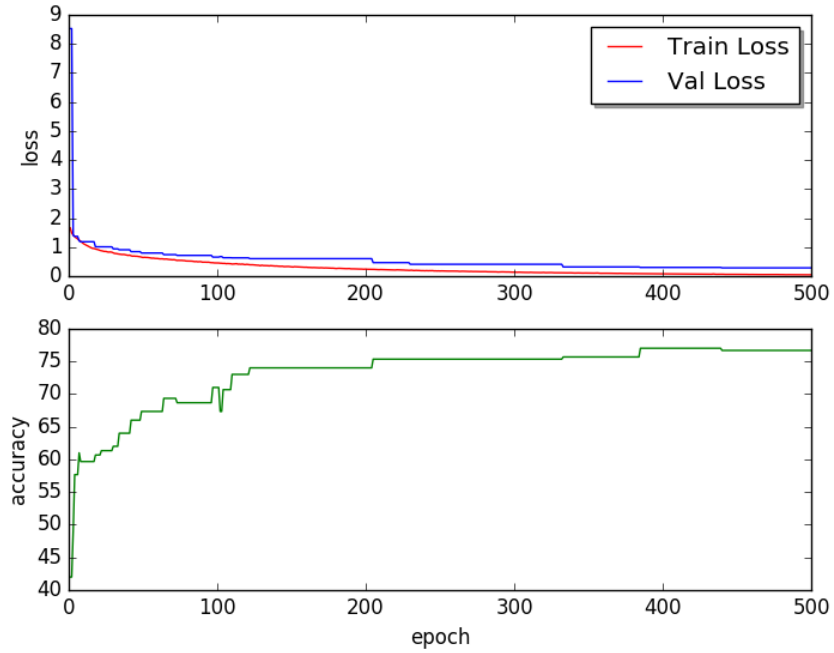

Abbildung 10.1: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 32x32 Pixeln und einer 9-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 213.372242928s
--- created class hammer in 177.402100801s
--- created class mug in 247.886315823s
--- created class scissor in 239.353957891s
--- created class headphone in 252.381113052s
--- created class teddy in 187.620648861s
Dataset generation took: 0h 21m 58s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training...
--- Epoch 1 of 500 took 2.18440198898s
--- test loss:           8.530048529307047
--- test accuracy rate:  42.000000178813934%
--- time left 0h 18m 12s
--- Epoch 50 of 500 took 1.5066010952s
--- test loss:           0.8028654555479685
--- test accuracy rate:  67.3333336909612%
--- time left 0h 11m 19s
--- Epoch 100 of 500 took 1.50862884521s
--- test loss:           0.6630017856756846
--- test accuracy rate:  70.9999995927016%
--- time left 0h 10m 4s
--- Epoch 150 of 500 took 1.50777387619s
--- test loss:           0.6093160212039948
--- test accuracy rate:  73.99999996026357%
--- time left 0h 8m 49s
--- Epoch 200 of 500 took 1.50746417046s
--- test loss:           0.6093160212039948
--- test accuracy rate:  73.99999996026357%
--- time left 0h 7m 33s
--- Epoch 250 of 500 took 1.51277208328s
--- test loss:           0.415282741189003
--- test accuracy rate:  75.33333351214728%
--- time left 0h 6m 19s
--- Epoch 300 of 500 took 1.50878500938s
--- test loss:           0.415282741189003
--- test accuracy rate:  75.33333351214728%
--- time left 0h 5m 3s
--- Epoch 350 of 500 took 1.51272106171s
--- test loss:           0.3238016838828723
--- test accuracy rate:  75.6666669001182%
--- time left 0h 3m 48s
--- Epoch 400 of 500 took 1.51198387146s
--- test loss:           0.30747852474451065
--- test accuracy rate:  77.00000057617822%
--- time left 0h 2m 32s
--- Epoch 450 of 500 took 1.51227092743s
--- test loss:           0.2924686248103778
--- test accuracy rate:  76.66666706403097%
--- time left 0h 1m 17s
--- Epoch 500 of 500 took 1.52149200439s
--- test loss:           0.2924686248103778
--- test accuracy rate:  76.66666706403097%
--- time left 0h 0m 1s
Final Accuracy: 76.66666706403097
Training took: 0h 14m 31s
```

AL BNN Framework mit 32x32 Pixeln und 11-lagiger Netzarchitektur

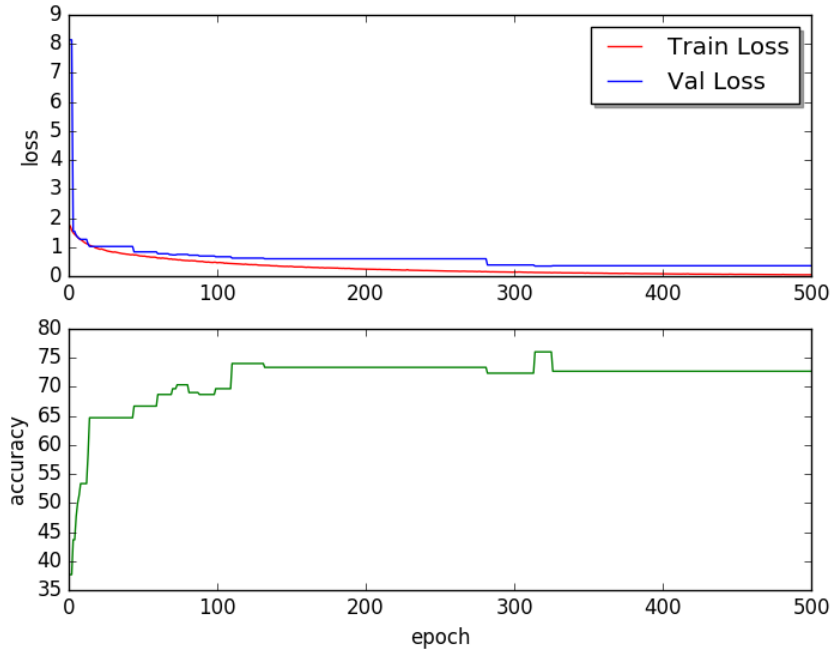

Abbildung 10.2: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 32x32 Pixeln und einer 11-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 213.372242928s
--- created class hammer in 177.402100801s
--- created class mug in 247.886315823s
--- created class scissor in 239.353957891s
--- created class headphone in 252.381113052s
--- created class teddy in 187.620648861s
Dataset generation took: 0h 21m 58s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training...
--- Epoch 1 of 500 took 3.13356184959s
--- test loss:           8.141099373499552
--- test accuracy rate:  37.66666650772095%
--- time left 0h 26m 6s
--- Epoch 50 of 500 took 2.20000195503s
--- test loss:           0.8417846063772837
--- test accuracy rate:  66.66666616996129%
--- time left 0h 16m 32s
--- Epoch 100 of 500 took 2.20698189735s
--- test loss:           0.674675722916921
--- test accuracy rate:  69.6666660408179%
--- time left 0h 14m 44s
--- Epoch 150 of 500 took 2.22544002533s
--- test loss:           0.6054696242014567
--- test accuracy rate:  73.33333293596904%
--- time left 0h 13m 1s
--- Epoch 200 of 500 took 2.23246002197s
--- test loss:           0.6054696242014567
--- test accuracy rate:  73.33333293596904%
--- time left 0h 11m 11s
--- Epoch 250 of 500 took 2.22815799713s
--- test loss:           0.6054696242014567
--- test accuracy rate:  73.33333293596904%
--- time left 0h 9m 19s
--- Epoch 300 of 500 took 2.23841404915s
--- test loss:           0.3919846365849177
--- test accuracy rate:  72.33333364129066%
--- time left 0h 7m 29s
--- Epoch 350 of 500 took 2.23231387138s
--- test loss:           0.3649934083223343
--- test accuracy rate:  72.66666665673256%
--- time left 0h 5m 37s
--- Epoch 400 of 500 took 2.22748184204s
--- test loss:           0.3649934083223343
--- test accuracy rate:  72.66666665673256%
--- time left 0h 3m 44s
--- Epoch 450 of 500 took 2.24138689041s
--- test loss:           0.3649934083223343
--- test accuracy rate:  72.66666665673256%
--- time left 0h 1m 54s
--- Epoch 500 of 500 took 2.23811197281s
--- test loss:           0.3649934083223343
--- test accuracy rate:  72.66666665673256%
--- time left 0h 0m 2s
Final Accuracy: 72.66666665673256
Training took: 0h 20m 36s
```

AL BNN Framework mit 32x32 Pixeln und 13-lagiger Netzarchitektur

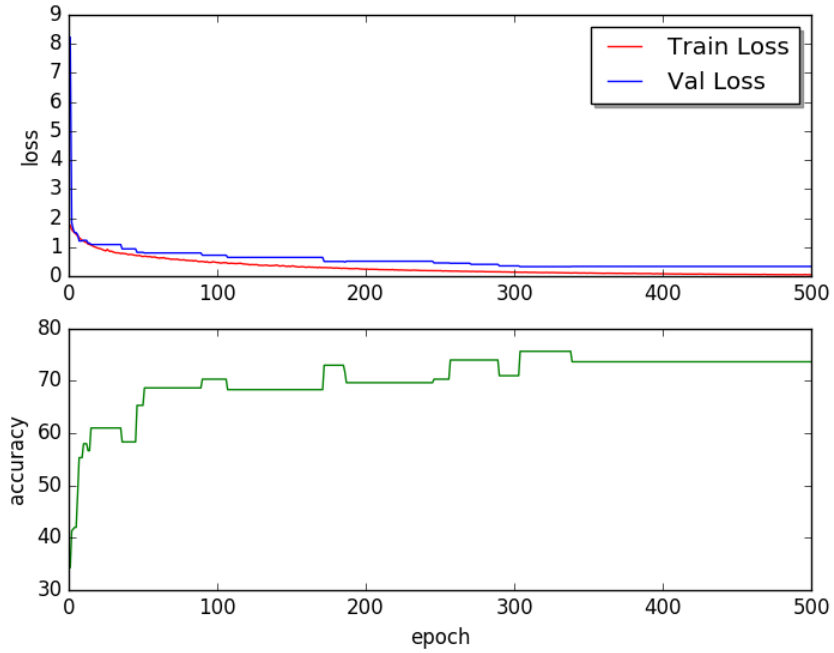

Abbildung 10.3: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 32x32 Pixeln und einer 13-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 213.372242928s
--- created class hammer in 177.402100801s
--- created class mug in 247.886315823s
--- created class scissor in 239.353957891s
--- created class headphone in 252.381113052s
--- created class teddy in 187.620648861s
Dataset generation took: 0h 21m 58s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
Training...
--- Epoch 1 of 500 took 4.15900111198s
--- test loss:           8.224351326624552
--- test accuracy rate:  34.33333337306976%
--- time left 0h 34m 39s
--- Epoch 50 of 500 took 2.94968914986s
--- test loss:           0.8252949317296346
--- test accuracy rate:  65.33333311478297%
--- time left 0h 22m 10s
--- Epoch 100 of 500 took 3.01456713676s
--- test loss:           0.7268659174442291
--- test accuracy rate:  70.33333331346512%
--- time left 0h 20m 8s
--- Epoch 150 of 500 took 2.9548368454s
--- test loss:           0.6526434024175009
--- test accuracy rate:  68.33333298563957%
--- time left 0h 17m 17s
--- Epoch 200 of 500 took 2.97296094894s
--- test loss:           0.5201890915632248
--- test accuracy rate:  69.66666628917058%
--- time left 0h 14m 54s
--- Epoch 250 of 500 took 2.9815659523s
--- test loss:           0.4617287864287694
--- test accuracy rate:  70.33333381017049%
--- time left 0h 12m 28s
--- Epoch 300 of 500 took 2.9833920002s
--- test loss:           0.3601759870847066
--- test accuracy rate:  71.00000058611234%
--- time left 0h 9m 59s
--- Epoch 350 of 500 took 2.99906396866s
--- test loss:           0.3411032383640607
--- test accuracy rate:  73.66666619976361%
--- time left 0h 7m 32s
--- Epoch 400 of 500 took 2.99399614334s
--- test loss:           0.3411032383640607
--- test accuracy rate:  73.66666619976361%
--- time left 0h 5m 2s
--- Epoch 450 of 500 took 2.9981842041s
--- test loss:           0.3411032383640607
--- test accuracy rate:  73.66666619976361%
--- time left 0h 2m 32s
--- Epoch 500 of 500 took 3.01012015343s
--- test loss:           0.3411032383640607
--- test accuracy rate:  73.66666619976361%
--- time left 0h 0m 3s
Final Accuracy: 73.66666619976361
Training took: 0h 27m 6s
```

AL BNN Framework mit 64x64 Pixeln und 9-lagiger Netzarchitektur

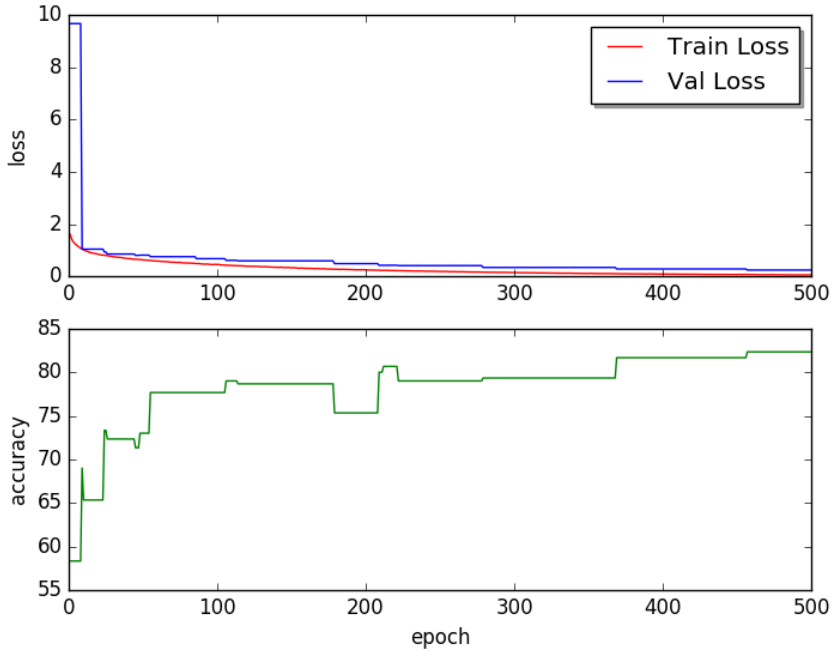

Abbildung 10.4: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 64x64 Pixeln und einer 9-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 250.454898119s
--- created class hammer in 185.149206161s
--- created class mug in 206.077868938s
--- created class scissor in 243.146188021s
--- created class headphone in 266.492260933s
--- created class teddy in 208.866183996s
Dataset generation took: 0h 22m 40s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training...
--- Epoch 1 of 500 took 5.04864382744s
--- test loss:           9.66779613494873
--- test accuracy rate:  58.33333383003871%
--- time left 0h 42m 4s
--- Epoch 50 of 500 took 3.6572599411s
--- test loss:           0.8113660911719004
--- test accuracy rate:  73.00000016887982%
--- time left 0h 27m 29s
--- Epoch 100 of 500 took 3.72813796997s
--- test loss:           0.677117109298706
--- test accuracy rate:  77.66666673123837%
--- time left 0h 24m 54s
--- Epoch 150 of 500 took 3.71836400032s
--- test loss:           0.5926511685053507
--- test accuracy rate:  78.66666701932749%
--- time left 0h 21m 45s
--- Epoch 200 of 500 took 3.70509195328s
--- test loss:           0.4864436239004135
--- test accuracy rate:  75.33333326379457%
--- time left 0h 18m 35s
--- Epoch 250 of 500 took 3.74805998802s
--- test loss:           0.41137540837128955
--- test accuracy rate:  79.00000015894571%
--- time left 0h 15m 40s
--- Epoch 300 of 500 took 3.74528479576s
--- test loss:           0.340203399459521
--- test accuracy rate:  79.33333329856396%
--- time left 0h 12m 32s
--- Epoch 350 of 500 took 3.73108100891s
--- test loss:           0.340203399459521
--- test accuracy rate:  79.33333329856396%
--- time left 0h 9m 23s
--- Epoch 400 of 500 took 3.72829389572s
--- test loss:           0.28338484714428586
--- test accuracy rate:  81.66666639347872%
--- time left 0h 6m 16s
--- Epoch 450 of 500 took 3.76073694229s
--- test loss:           0.28338484714428586
--- test accuracy rate:  81.66666639347872%
--- time left 0h 3m 11s
--- Epoch 500 of 500 took 3.76583099365s
--- test loss:           0.2410653680562973
--- test accuracy rate:  82.3333335419496%
--- time left 0h 0m 3s
Final Accuracy: 82.3333335419496
Training took: 0h 33m 4s
```

AL BNN Framework mit 64x64 Pixeln und 11-lagiger Netzarchitektur

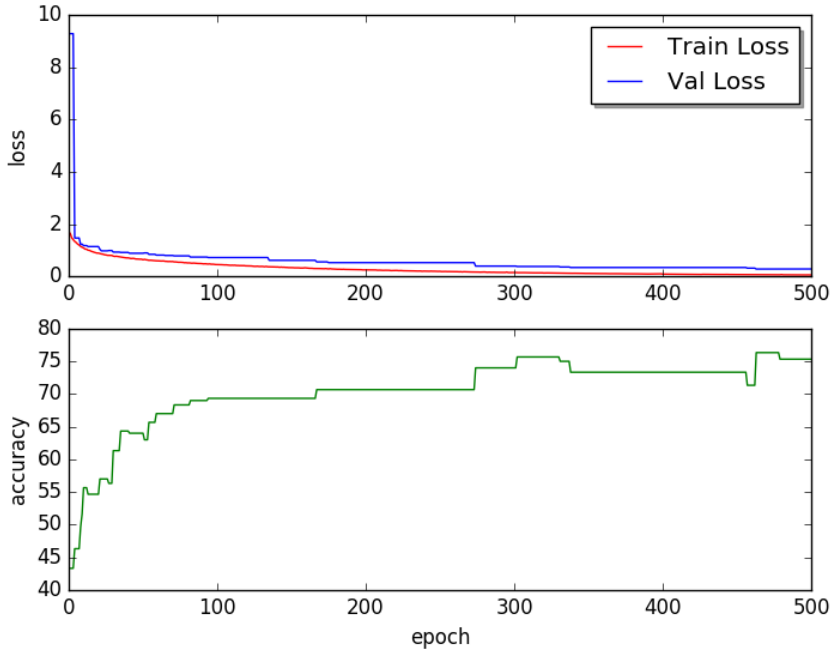

Abbildung 10.5: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 64x64 Pixeln und einer 11-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 250.454898119s
--- created class hammer in 185.149206161s
--- created class mug in 206.077868938s
--- created class scissor in 243.146188021s
--- created class headphone in 266.492260933s
--- created class teddy in 208.866183996s
Dataset generation took: 0h 22m 40s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training...
--- Epoch 1 of 500 took 7.43483901024s
--- test loss:           9.281754811604818
--- test accuracy rate:  43.33333273728689%
--- time left 1h 1m 57s
--- Epoch 50 of 500 took 6.37737011909s
--- test loss:           0.8829614520072937
--- test accuracy rate:  64.00000005960464%
--- time left 0h 47m 56s
--- Epoch 100 of 500 took 6.42109298706s
--- test loss:           0.7199085156122843
--- test accuracy rate:  69.33333327372868%
--- time left 0h 42m 54s
--- Epoch 150 of 500 took 6.45780992508s
--- test loss:           0.6131301323572794
--- test accuracy rate:  69.33333327372868%
--- time left 0h 37m 46s
--- Epoch 200 of 500 took 6.51446604729s
--- test loss:           0.5245678375164667
--- test accuracy rate:  70.66666682561238%
--- time left 0h 32m 40s
--- Epoch 250 of 500 took 6.47503113747s
--- test loss:           0.5245678375164667
--- test accuracy rate:  70.66666682561238%
--- time left 0h 27m 5s
--- Epoch 300 of 500 took 6.54991793633s
--- test loss:           0.39275391896565753
--- test accuracy rate:  73.99999996026357%
--- time left 0h 21m 56s
--- Epoch 350 of 500 took 6.52018713951s
--- test loss:           0.33828331530094147
--- test accuracy rate:  73.33333343267441%
--- time left 0h 16m 24s
--- Epoch 400 of 500 took 6.51181197166s
--- test loss:           0.33828331530094147
--- test accuracy rate:  73.33333343267441%
--- time left 0h 10m 57s
--- Epoch 450 of 500 took 6.5406730175s
--- test loss:           0.33828331530094147
--- test accuracy rate:  73.33333343267441%
--- time left 0h 5m 33s
--- Epoch 500 of 500 took 6.56290102005s
--- test loss:           0.28229610870281857
--- test accuracy rate:  75.33333351214728%
--- time left 0h 0m 6s
Final Accuracy: 75.33333351214728
Training took: 0h 56m 23s
```

AL BNN Framework mit 64x64 Pixeln und 13-lagiger Netzarchitektur

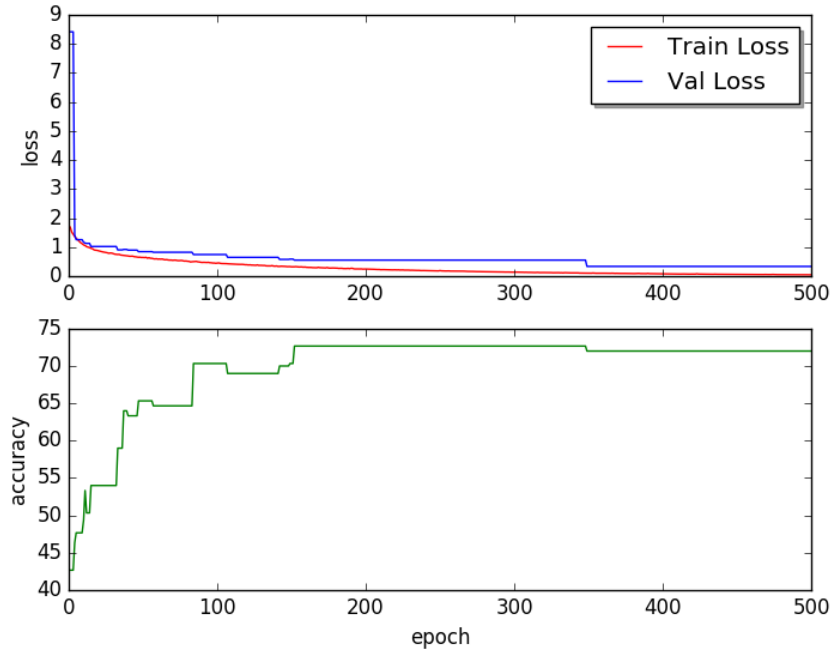

Abbildung 10.6: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 64x64 Pixeln und einer 13-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 250.454898119s
--- created class hammer in 185.149206161s
--- created class mug in 206.077868938s
--- created class scissor in 243.146188021s
--- created class headphone in 266.492260933s
--- created class teddy in 208.866183996s
Dataset generation took: 0h 22m 40s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
Training...
--- Epoch 1 of 500 took 9.70549893379s
--- test loss:           8.416812578837076
--- test accuracy rate:  42.66666720310847%
--- time left 1h 20m 52s
--- Epoch 50 of 500 took 7.80850100517s
--- test loss:           0.8501531779766083
--- test accuracy rate:  65.33333336313565%
--- time left 0h 58m 41s
--- Epoch 100 of 500 took 7.87756705284s
--- test loss:           0.7510710755983988
--- test accuracy rate:  70.33333306511243%
--- time left 0h 52m 38s
--- Epoch 150 of 500 took 7.8083782196s
--- test loss:           0.59305273493131
--- test accuracy rate:  70.33333306511243%
--- time left 0h 45m 40s
--- Epoch 200 of 500 took 7.93614912033s
--- test loss:           0.5590745160977045
--- test accuracy rate:  72.66666690508525%
--- time left 0h 39m 48s
--- Epoch 250 of 500 took 7.92737698555s
--- test loss:           0.5590745160977045
--- test accuracy rate:  72.66666690508525%
--- time left 0h 33m 9s
--- Epoch 300 of 500 took 7.98374986649s
--- test loss:           0.5590745160977045
--- test accuracy rate:  72.66666690508525%
--- time left 0h 26m 44s
--- Epoch 350 of 500 took 7.87011194229s
--- test loss:           0.34282491107781726
--- test accuracy rate:  71.99999988079071%
--- time left 0h 19m 48s
--- Epoch 400 of 500 took 7.98774003983s
--- test loss:           0.3412327667077382
--- test accuracy rate:  71.99999988079071%
--- time left 0h 13m 26s
--- Epoch 450 of 500 took 8.02343797684s
--- test loss:           0.3412327667077382
--- test accuracy rate:  71.99999988079071%
--- time left 0h 6m 49s
--- Epoch 500 of 500 took 8.0839240551s
--- test loss:           0.3412327667077382
--- test accuracy rate:  71.99999988079071%
--- time left 0h 0m 8s
Final Accuracy: 71.99999988079071
Training took: 1h 8m 33s
```

AL BNN Framework mit 128x128 Pixeln und 9-lagiger Netzarchitektur

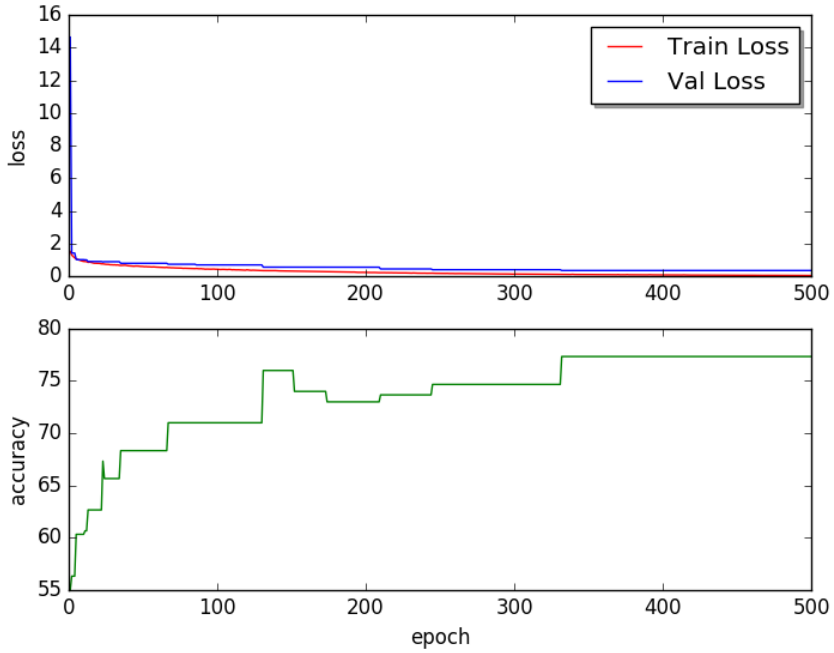

Abbildung 10.7: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 128x128 Pixeln und einer 9-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 271.110031843s
--- created class hammer in 236.607131958s
--- created class mug in 261.389991999s
--- created class scissor in 263.375129938s
--- created class headphone in 287.739795923s
--- created class teddy in 202.677869081s
Dataset generation took: 0h 25m 22s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
Training...
--- Epoch 1 of 500 took 17.1315469742s
--- test loss:           14.634090582529703
--- test accuracy rate:  55.00000069538753%
--- time left 2h 22m 45s
--- Epoch 50 of 500 took 13.0510830879s
--- test loss:           0.7993665238221487
--- test accuracy rate:  68.33333373069763%
--- time left 1h 38m 6s
--- Epoch 100 of 500 took 13.2174329758s
--- test loss:           0.7004175881544749
--- test accuracy rate:  70.99999984105428%
--- time left 1h 28m 20s
--- Epoch 150 of 500 took 13.249753952s
--- test loss:           0.5642440319061279
--- test accuracy rate:  76.00000028808913%
--- time left 1h 17m 30s
--- Epoch 200 of 500 took 13.2743170261s
--- test loss:           0.5593559046586355
--- test accuracy rate:  72.99999992052715%
--- time left 1h 6m 35s
--- Epoch 250 of 500 took 13.3365781307s
--- test loss:           0.40598701934019726
--- test accuracy rate:  74.66666648785274%
--- time left 0h 55m 47s
--- Epoch 300 of 500 took 13.2536659241s
--- test loss:           0.40598701934019726
--- test accuracy rate:  74.66666648785274%
--- time left 0h 44m 23s
--- Epoch 350 of 500 took 13.2155048847s
--- test loss:           0.35833404461542767
--- test accuracy rate:  77.3333334674438%
--- time left 0h 33m 15s
--- Epoch 400 of 500 took 13.2666108608s
--- test loss:           0.35833404461542767
--- test accuracy rate:  77.3333334674438%
--- time left 0h 22m 19s
--- Epoch 450 of 500 took 13.2681419849s
--- test loss:           0.35833404461542767
--- test accuracy rate:  77.3333334674438%
--- time left 0h 11m 16s
--- Epoch 500 of 500 took 13.345209837s
--- test loss:           0.35833404461542767
--- test accuracy rate:  77.3333334674438%
--- time left 0h 0m 13s
Final Accuracy: 77.3333334674438
Training took: 1h 53m 14s
```

AL BNN Framework mit 128x128 Pixeln und 11-lagiger Netzarchitektur

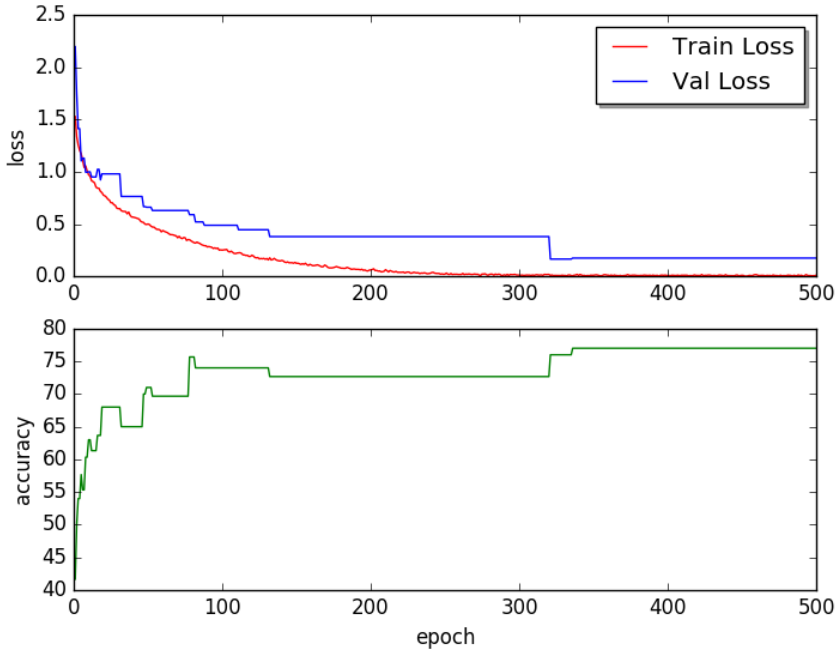

Abbildung 10.8: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 128x128 Pixeln und einer 11-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 271.110031843s
--- created class hammer in 236.607131958s
--- created class mug in 261.389991999s
--- created class scissor in 263.375129938s
--- created class headphone in 287.739795923s
--- created class teddy in 202.677869081s
Dataset generation took: 0h 25m 22s
Training...
--- Epoch 1 of 500 took 27.6335339546s
--- test loss:           2.1959631542364755
--- test accuracy rate:  41.666666666666664%
--- time left 3h 50m 16s
--- Epoch 50 of 500 took 25.0742008686s
--- test loss:           0.6610519289970398
--- test accuracy rate:  71.00000033775966%
```

```
--- time left 3h 8m 28s
--- Epoch 100 of 500 took 25.4720880985s
--- test loss:           0.48843809713919956
--- test accuracy rate:  74.00000020861626%
--- time left 2h 50m 14s
--- Epoch 150 of 500 took 25.5441451073s
--- test loss:           0.3800725390513738
--- test accuracy rate:  72.66666665673256%
--- time left 2h 29m 25s
--- Epoch 200 of 500 took 25.9456791878s
--- test loss:           0.3800725390513738
--- test accuracy rate:  72.66666665673256%
--- time left 2h 10m 9s
--- Epoch 250 of 500 took 25.5431690216s
--- test loss:           0.3800725390513738
--- test accuracy rate:  72.66666665673256%
--- time left 1h 46m 51s
--- Epoch 300 of 500 took 25.5628349781s
--- test loss:           0.3800725390513738
--- test accuracy rate:  72.66666665673256%
--- time left 1h 25m 38s
--- Epoch 350 of 500 took 25.5358691216s
--- test loss:           0.17464689258486032
--- test accuracy rate:  77.00000014156103%
--- time left 1h 4m 15s
--- Epoch 400 of 500 took 25.4881160259s
--- test loss:           0.17464689258486032
--- test accuracy rate:  77.00000014156103%
--- time left 0h 42m 54s
--- Epoch 450 of 500 took 25.3461608887s
--- test loss:           0.17464689258486032
--- test accuracy rate:  77.00000014156103%
--- time left 0h 21m 32s
--- Epoch 500 of 500 took 25.4208059311s
--- test loss:           0.17464689258486032
--- test accuracy rate:  77.00000014156103%
--- time left 0h 0m 25s
Final Accuracy: 77.00000014156103
Training took: 3h 35m 51s
```

AL BNN Framework mit 128x128 Pixeln und 13-lagiger Netzarchitektur

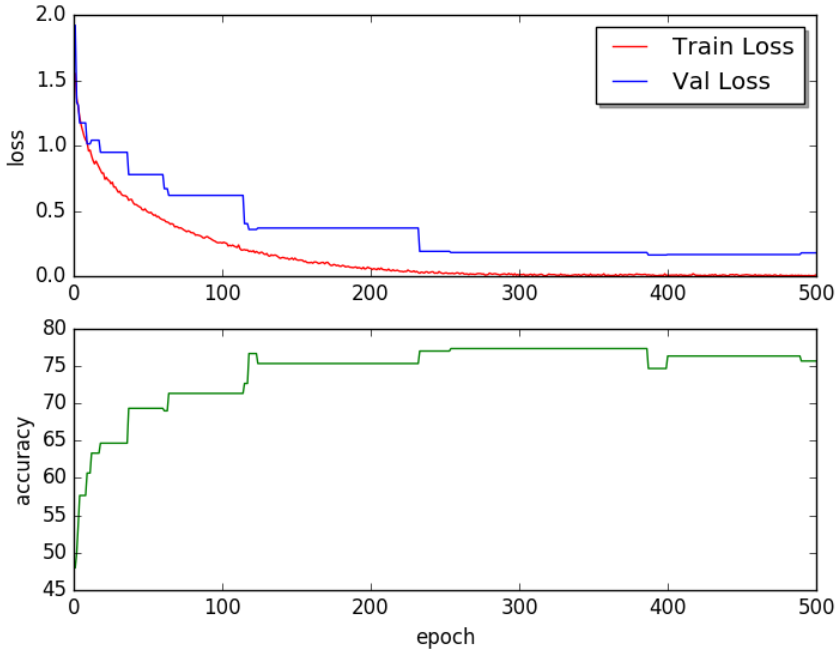

Abbildung 10.9: Trainingsverlauf mit dem AL BNN Framework des Binären Neuronalen Netzes mit einer Eingangsauflösung von 128x128 Pixeln und einer 13-lagigen Netzarchitektur

```
Using gpu device 0: GeForce GTX 970 (CNMeM is enabled with initial
    size: 80.0% of memory, cuDNN 5110)
Creating Dataset ...
--- created class bottle in 271.110031843s
--- created class hammer in 236.607131958s
--- created class mug in 261.389991999s
--- created class scissor in 263.375129938s
--- created class headphone in 287.739795923s
--- created class teddy in 202.677869081s
Dataset generation took: 0h 25m 22s
Training ...
Loading AL dataset...
CLASS LABELS: ['bottle', 'hammer', 'mug', 'scissor', 'headphone', '
    teddy']
TRAINING IMAGES: 1350
VALIDATION IMAGES: 150
TEST IMAGES: 300
Building the BNN...
```

```
Training...
--- Epoch 1 of 500 took 35.1319971085s
--- test loss:           1.919224629799525
--- test accuracy rate:  48.00000041723251%
--- time left 4h 52m 45s
--- Epoch 50 of 500 took 30.2951478958s
--- test loss:           0.7785362253586451
--- test accuracy rate:  69.33333339790504%
--- time left 3h 47m 43s
--- Epoch 100 of 500 took 30.4887621403s
--- test loss:           0.6193165481090546
--- test accuracy rate:  71.33333360155423%
--- time left 3h 23m 45s
--- Epoch 150 of 500 took 30.6147282124s
--- test loss:           0.36947496235370636
--- test accuracy rate:  75.33333338797092%
--- time left 2h 59m 5s
--- Epoch 200 of 500 took 30.6215150356s
--- test loss:           0.36947496235370636
--- test accuracy rate:  75.33333338797092%
--- time left 2h 33m 37s
--- Epoch 250 of 500 took 30.6023230553s
--- test loss:           0.191284808019797
--- test accuracy rate:  77.00000032782555%
--- time left 2h 8m 1s
--- Epoch 300 of 500 took 30.6336269379s
--- test loss:           0.18239254939059416
--- test accuracy rate:  77.3333334053556%
--- time left 1h 42m 37s
--- Epoch 350 of 500 took 30.5517120361s
--- test loss:           0.18239254939059416
--- test accuracy rate:  77.3333334053556%
--- time left 1h 16m 53s
--- Epoch 400 of 500 took 35.7260811329s
--- test loss:           0.16683158775170645
--- test accuracy rate:  76.33333330353102%
--- time left 1h 0m 8s
--- Epoch 450 of 500 took 30.6692109108s
--- test loss:           0.16683158775170645
--- test accuracy rate:  76.33333330353102%
--- time left 0h 26m 4s
--- Epoch 500 of 500 took 30.6991610527s
--- test loss:           0.17981350918610892
--- test accuracy rate:  75.66666677594185%
--- time left 0h 0m 30s
Final Accuracy: 75.66666677594185
Training took: 4h 17m 42s
```